
Programming Heterogeneous (GPU) Systems

Jeffrey Vetter

Presented to
Extreme Scale Computing Training Program
ANL: St. Charles, IL
2 August 2013

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

**Georgia
Tech**  **College of
Computing**
Computational Science and Engineering

<http://ft.ornl.gov> ♦ vetter@computer.org

Keeneland

Home | Login

COMPUTING SYSTEMSABOUTSOFTWAREPUBLICATIONS/PRESENTATIONSEDUCATIONAL MATERIALSNEWSACKNOWLEDGEMENTS

Quick Links





- Quick Start Guide
- Batch Scripts
- New Account
- Compiling
- Scheduling
- Remote Desktop Using NX
- Keeneland Project
- KFS (XSEDE)
- KIDS Home

Courses

Take the free, online, interactive, student paced, GPU course from Udacity!

Intro to Parallel Programming
(Udacity CS344)

Keeneland Partners



Search

Search this site:

Home

Quick Start Guide

- System Overview
- Getting a NICS Account
- Getting Help
- Logging In
- Configuring your Environment
 - Modules
 - Notes
- File Systems and Storage
- Software Development
 - Compilers
 - CUDA
 - OpenCL
 - MPI
 - Control Version Systems
- Running Jobs
 - Batch Jobs
 - Notes on Batch Scripts
 - NUMA
 - Launching Jobs
 - Queues
 - Output
- Known Problems

System Overview

The Keeneland Initial Delivery (KID) system, which was delivered in October 2010. It is composed of an HP SL-390 (Ariston) cluster with Intel Westmere hex-core CPUs, NVIDIA 6GB Fermi GPUs, and a Qlogic QDR InfiniBand interconnect. Each node has two hex-core CPUs and 3 GPUs, with a total of 120 nodes, 240 CPUs and 360 GPUs.

Getting a NICS Account

Please see [Getting Access to Keeneland](#) for details on getting an account.

Once you have an account, you will be added to the [Keeneland Users](#) mailing list. System-wide announcements will broadcast to this list.

Getting Help

Please direct any questions to help@xsede.org. To ensure your question gets routed correctly, please include "Keeneland" in the subject line.

Tutorial accounts use "UT-NTNLEDU" for an allocation in the job scheduler

The Scalable Heterogeneous Computing (SHOC) Benchmark Suite

PI: Jeffrey S. Vetter, ORNL
Future Technologies Group

<https://github.com/vetter/shoc>

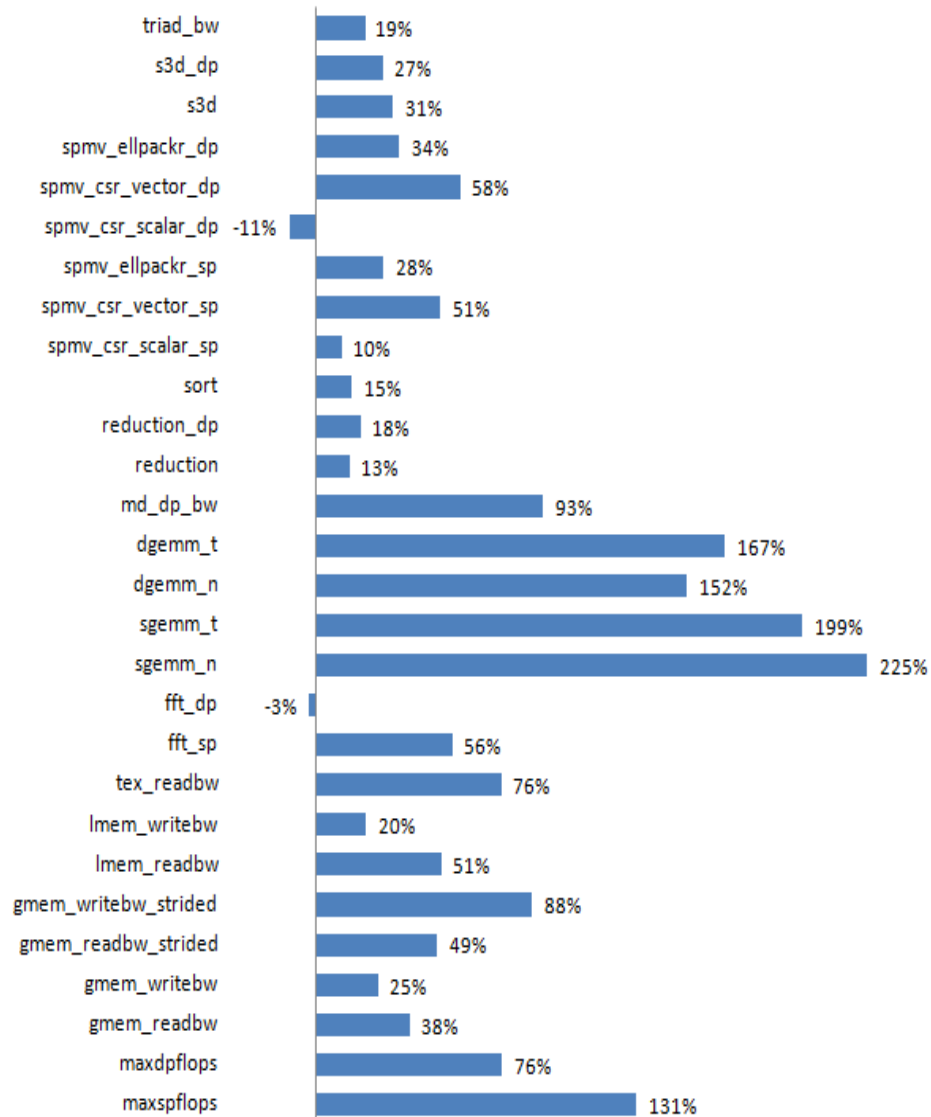
Objectives

- Design and implement a set of performance and stability tests for HPC systems with heterogeneous architectures
- Implemented each test in MPI, OpenCL, CUDA to
 - Evaluate the differences in these emerging programming models
 - MIC to be released shortly
 - OpenACC coming soon
- Sponsored by NSF, DOE

Accomplishments

- Consistent open source software releases
 - Over 10000 downloads internationally since 2010
 - Used in multiple procurements worldwide
 - Used by vendors and researchers for testing, understanding
- Across diverse range of architectures: NVIDIA, AMD, ARM, Intel, even Android
- Overview published at 3rd Workshop General-Purpose Computation on Graphics Processing Units (GPGPU '10): ~100 citations to date

A. Danalis, G. Marin, C. McCurdy, J. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, and J.S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in Third Workshop on General-Purpose Computation on Graphics Processors (GPGPU 2010). Pittsburgh, 2010



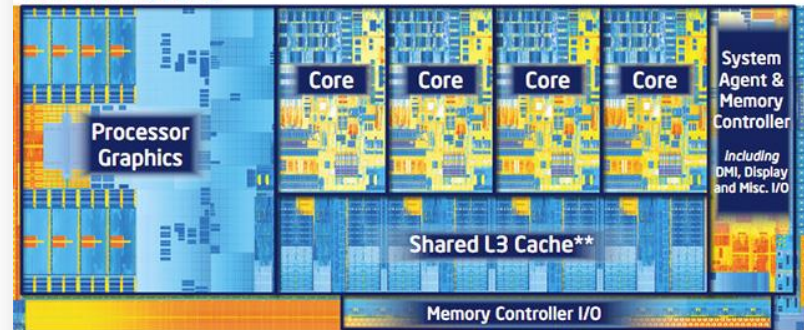
This chart shows the "out of the box" improvement from NVIDIA Fermi (M2090) to Kepler (K20m). Measured using CUDA 5.0 with an identical host system. Largest improvements observed in compute intensive workloads. Modest increases for memory bound kernels. No increase in DP FFT, suggests CUFFT not completely optimized for Kepler in release 5.0.

Motivation

Emerging Computing Architectures

- Heterogeneous processing
 - Many cores
 - Fused, configurable memory
- Memory
 - 3D Stacking
 - New devices (PCRAM, ReRAM)
- Interconnects
 - Collective offload
 - Scalable topologies
- Storage
 - Active storage
 - Non-traditional storage architectures (key-value stores)
- Improving performance and programmability in face of increasing complexity
 - Power, resilience

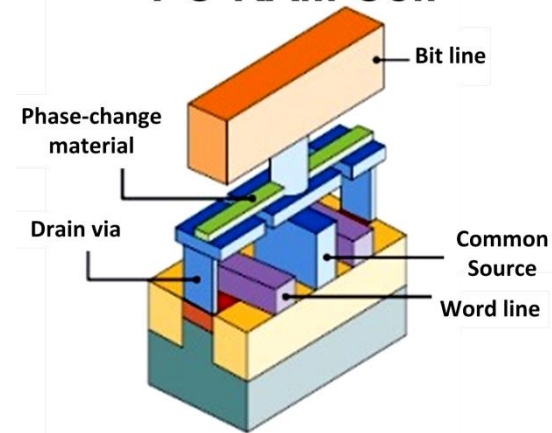
3rd Generation Intel® Core™ Processor:
22nm Process



New architecture with shared cache delivering more performance and energy efficiency

Quad Core die with Intel® HD Graphics 4000 shown above
Transistor count: 1.4Billion Die size: 160mm²
** Cache is shared across all 4 cores and processor graphics

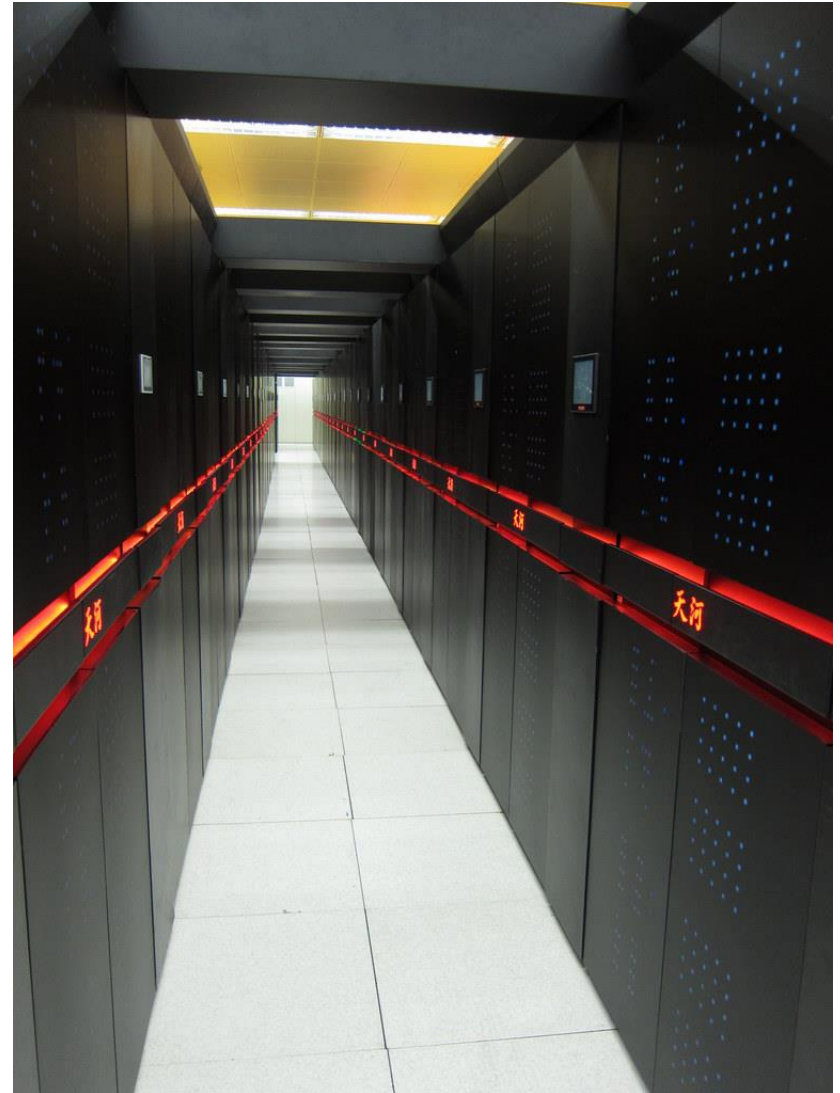
PC-RAM Cell



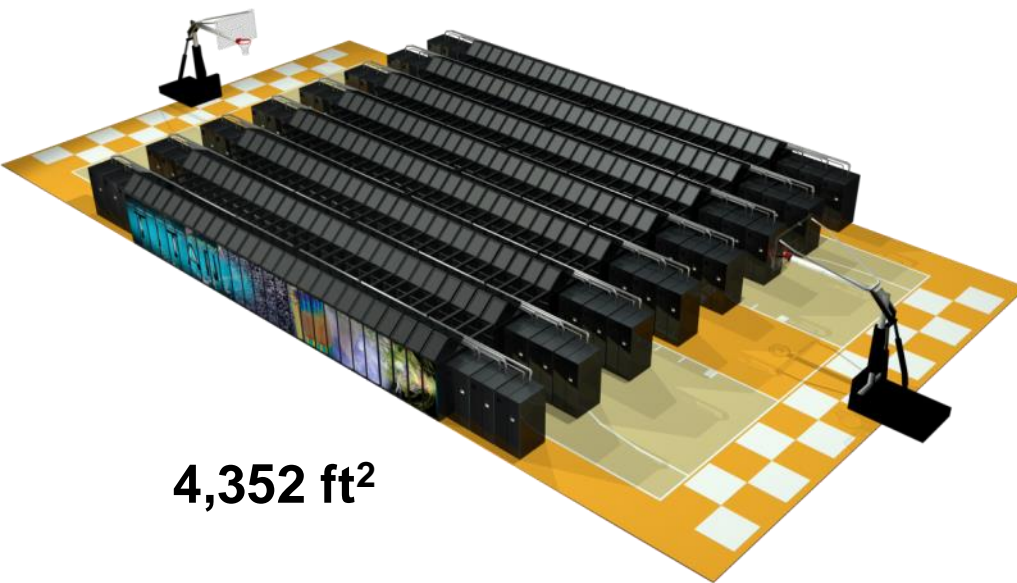
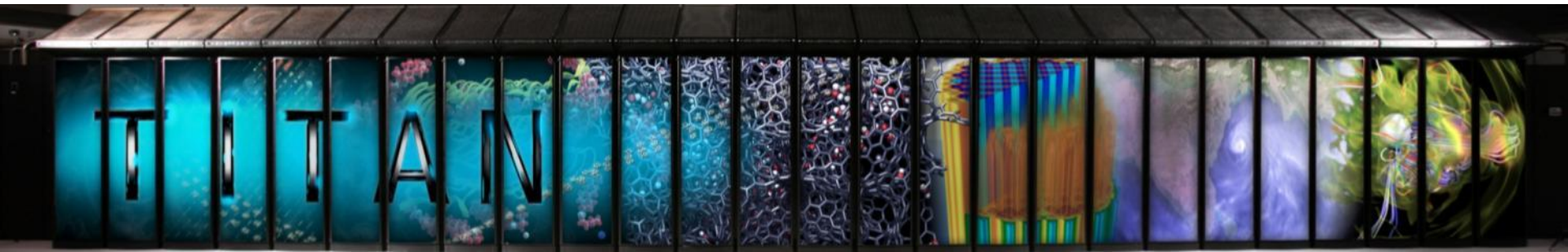
HPC (all) computer design is more fluid now than in the past two decades.

TH-2 System

- **Compute Nodes have 3.432 Tflop/s per node**
 - 16,000 nodes
 - 32000 Intel Xeon cpus
 - 48000 Intel Xeon phis
- **Operations Nodes**
 - 4096 FT CPUs as operations nodes
- **Proprietary interconnect TH2 express**
- **1PB memory (host memory only)**
- **Global shared parallel storage is 12.4 PB**
- **Cabinets: $125+13+24 = 162$ compute/communication/storage cabinets**
 - ~750 m²
- **NUDT and Inspur**



ORNL's "Titan" Hybrid System: Cray XK7 with AMD Opteron and NVIDIA Tesla processors



4,352 ft²

SYSTEM SPECIFICATIONS:

- Peak performance of 27.1 PF
 - 24.5 GPU + 2.6 CPU
- 18,688 Compute Nodes each with:
 - 16-Core AMD Opteron CPU
 - NVIDIA Tesla "K20x" GPU
 - 32 + 6 GB memory
- 512 Service and I/O nodes
- 200 Cabinets
- 710 TB total system memory
- Cray Gemini 3D Torus Interconnect
- 8.9 MW peak power

Contemporary HPC Architectures

Date	System	Location	Comp	Comm	Peak (PF)	Power (MW)
2009	Jaguar; Cray XT5	ORNL	AMD 6c	Seastar2	2.3	7.0
2010	Tianhe-1A	NSC Tianjin	Intel + NVIDIA	Proprietary	4.7	4.0
2010	Nebulae	NSCS Shenzhen	Intel + NVIDIA	IB	2.9	2.6
2010	Tsubame 2	TiTech	Intel + NVIDIA	IB	2.4	1.4
2011	K Computer	RIKEN/Kobe	SPARC64 VIIIfx	Tofu	10.5	12.7
2012	Titan; Cray XK6	ORNL	AMD + NVIDIA	Gemini	10-20	9
2012	Mira; BlueGeneQ	ANL	SoC	Proprietary	10	3.9
2012	Sequoia; BlueGeneQ	LLNL	SoC	Proprietary	20	7.9
2012	Blue Waters; Cray	NCSA/UIUC	AMD + (partial) NVIDIA	Gemini	11.6	
2013	Stampede	TACC	Intel + MIC	IB	9.5	5
2013	Tianhe-2	NSCC-GZ (Guangzhou)	Intel + MIC	Proprietary	54	~20

AMD Llano's fused memory hierarchy

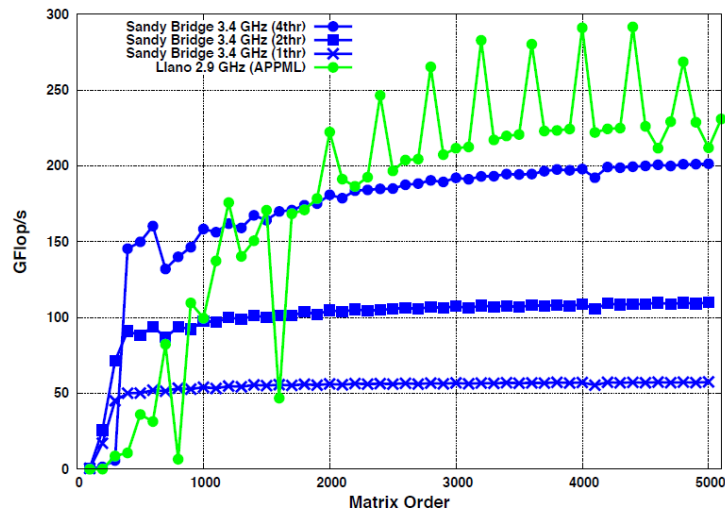
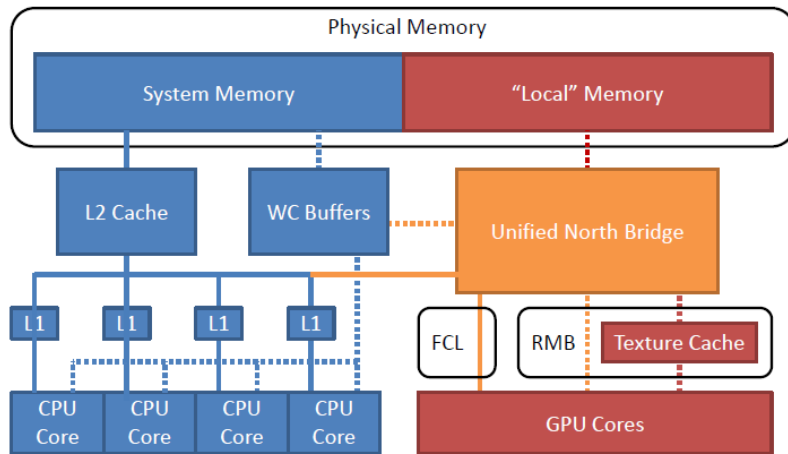
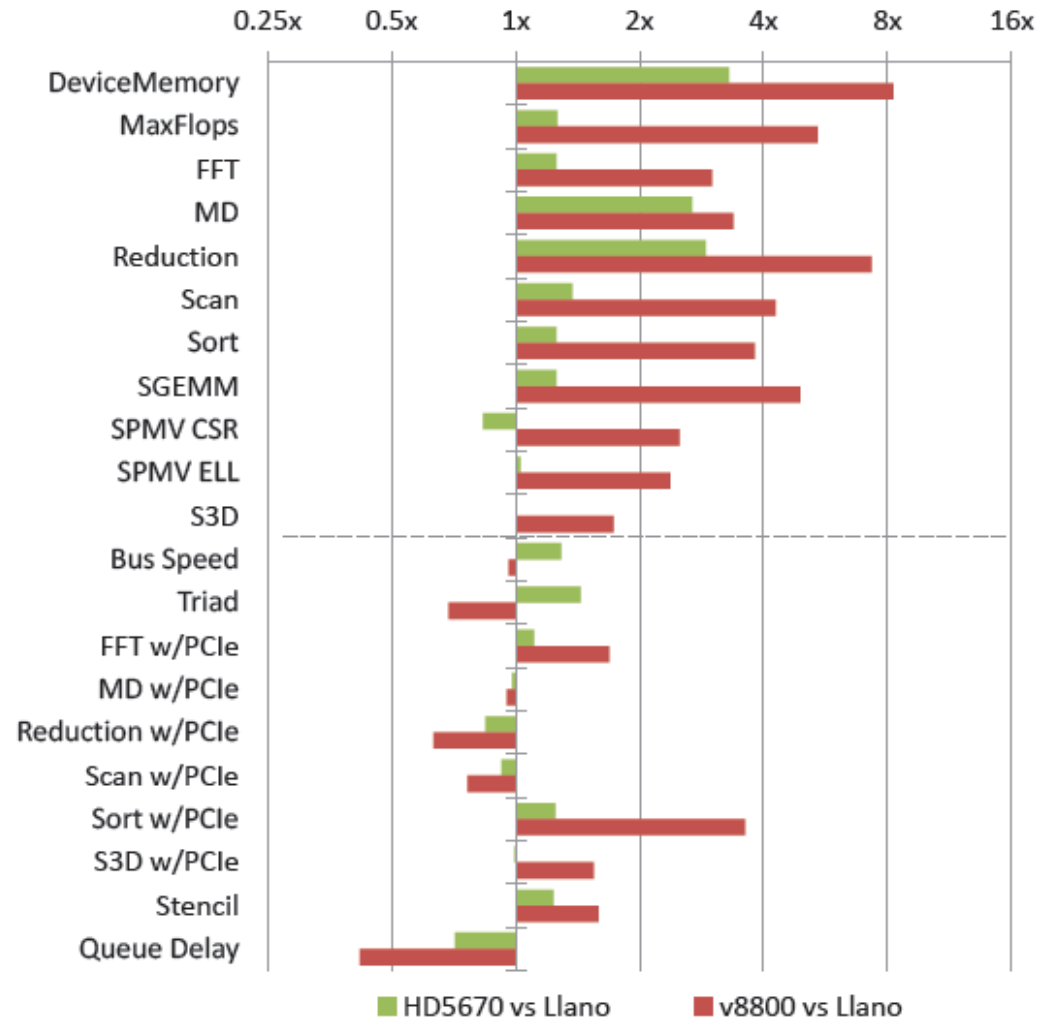


Figure 3: SGEMM Performance (one, two, and four CPU threads for Sandy Bridge and the OpenCL-based AMD APPML for Llano's fGPU)

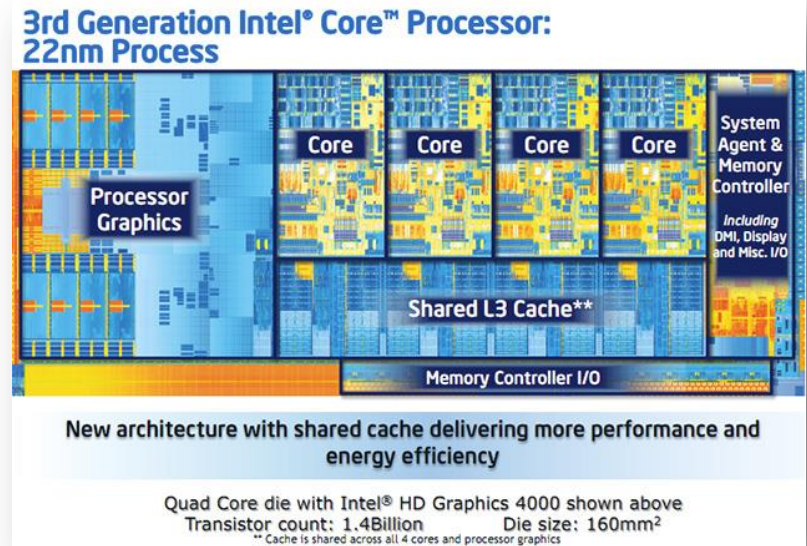


K. Spafford, J.S. Meredith, S. Lee, D. Li, P.C. Roth, and J.S. Vetter, "The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Architectures," in ACM Computing Frontiers (CF). Cagliari, Italy: ACM, 2012.

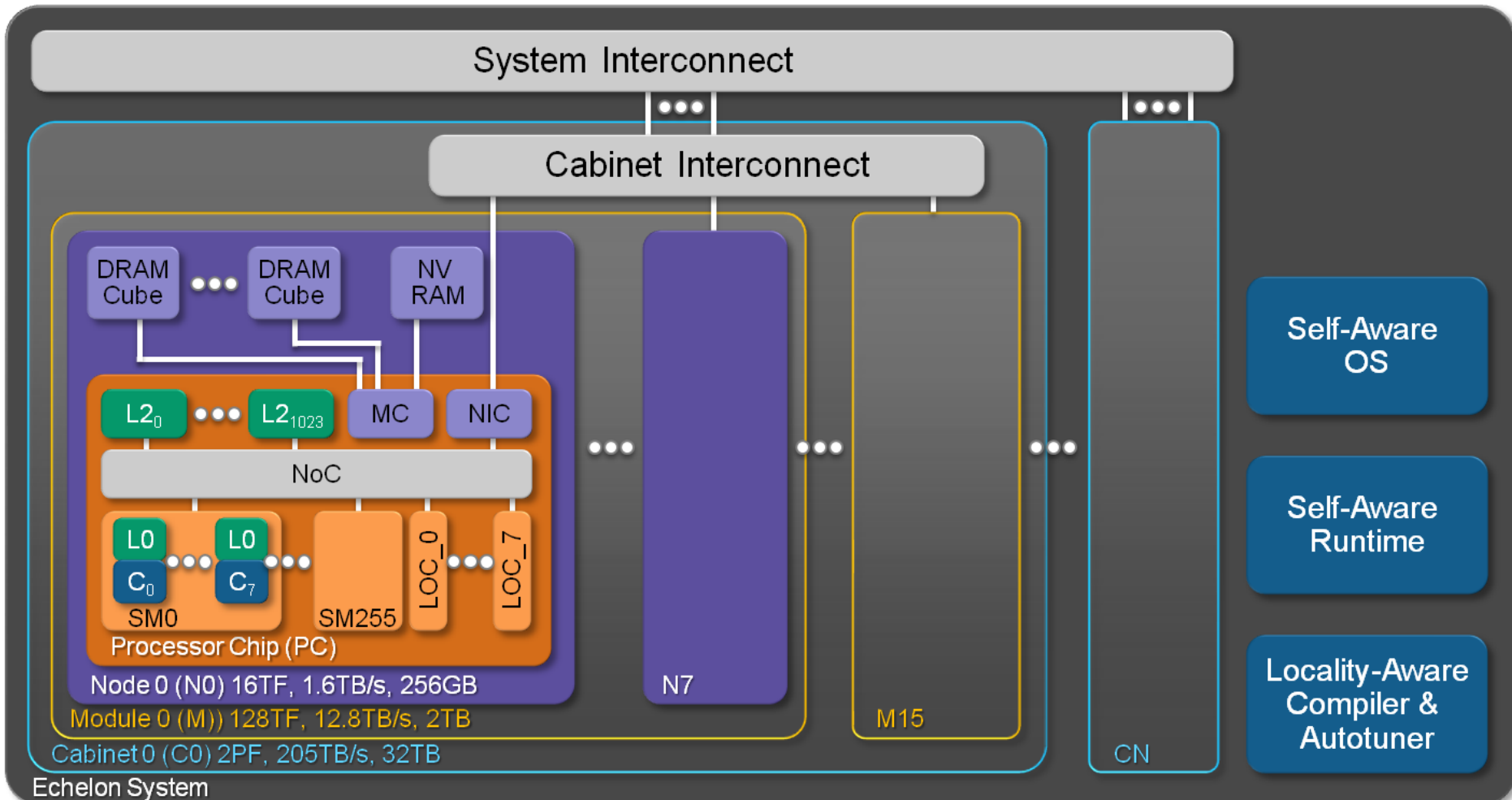
Note: Both SB and Llano are consumer parts, not server parts.

Future Directions in Heterogeneous Computing

- Over the next decade: Heterogeneous computing will continue to increase in importance
- Manycore
- Hardware features
 - Transactional memory
 - Random Number Generators
 - Scatter/Gather
 - Wider SIMD/AVX
- Synergies with BIGDATA, mobile markets, graphics
- Top 10 list of features to include *from application perspective.*
Now is the time!



NVIDIA Echelon System Sketch



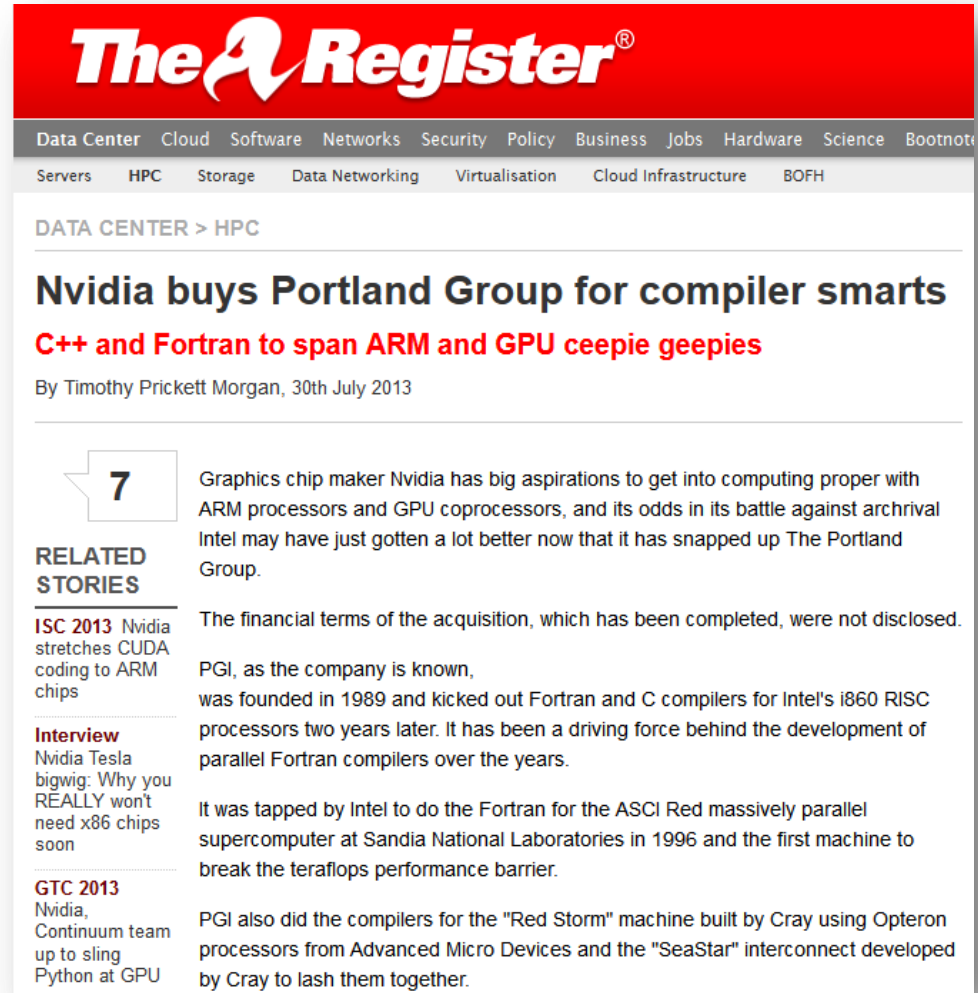
NVIDIA Echelon team: NVIDIA, ORNL, Micron, Cray, Georgia Tech, Stanford, UC-Berkeley, U Penn, Utah, Tennessee, Lockheed Martin

DARPA HPC Funded Project

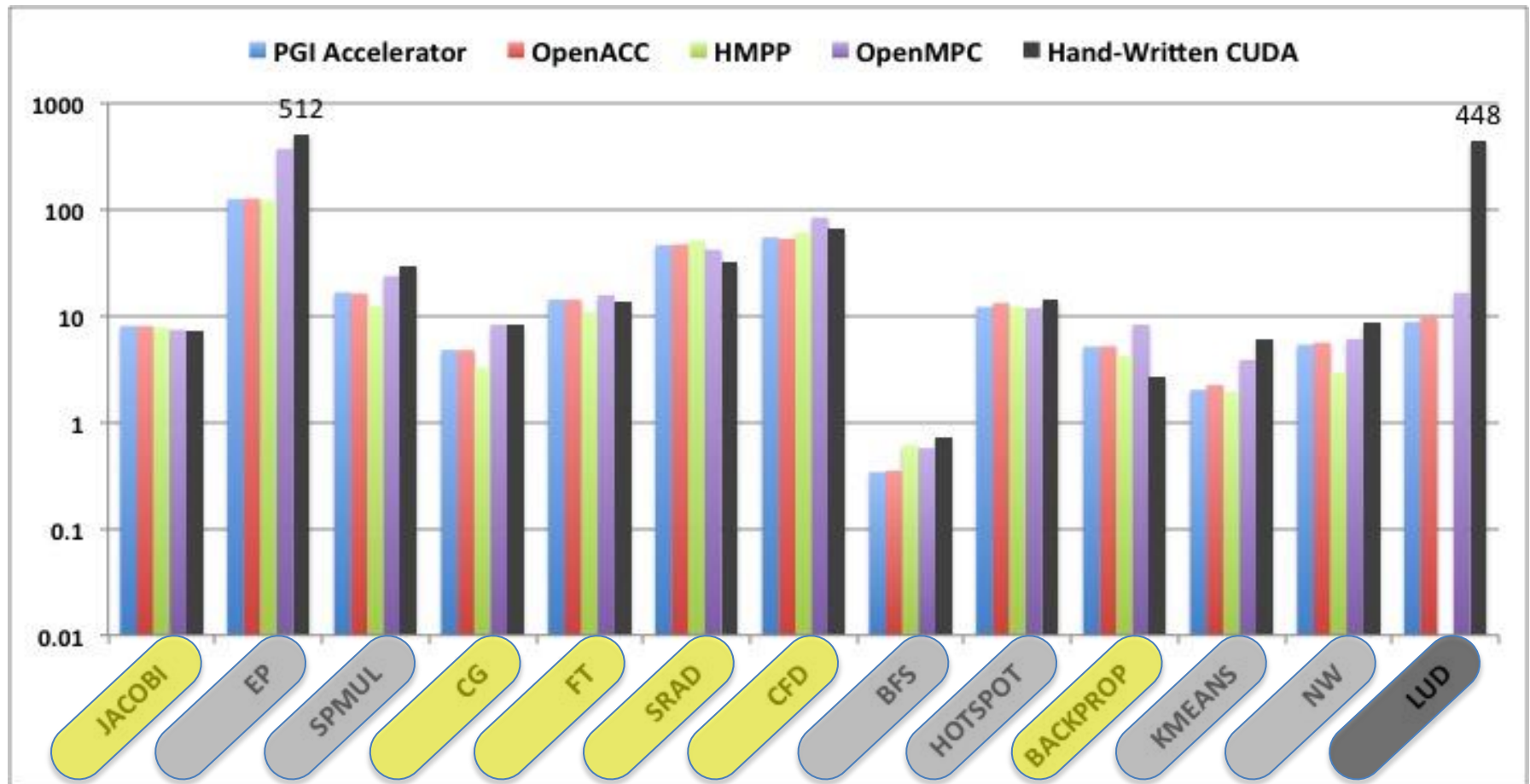


Critical Implications for Software, Apps, Developers

- Functional portability
- Performance portability
- Fast moving research, standards, products
- Incompatibilities among models
- Rewrite your code every 5 years
- Jobs!



Performance of Directive-based GPU Programming Models Gaining on Hand-Written CUDA



•Speedups are over serial on the CPU compiled with GCC v4.1.2 using option -O3, when the largest available input data were used.

•Experimental Platform: CPU: Intel Xeon at 2.8 GHz GPU: NVIDIA Tesla M2090 with 512 CUDA cores at 1.15GHz

Keeneland Overview



Keeneland – Full Scale System

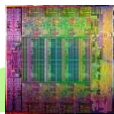
Initial Delivery system installed in Oct 2010

- 201 TFLOPS in 7 racks (90 sq ft incl service area)
- 902 MFLOPS per watt on HPL (#12 on Green500)
- Upgraded April 2012 to 255 TFLOPS
- Over 200 users, 100 projects using KID

Full scale system installed in Oct 2012

- 792 M2090 GPUs contribute to aggregate system peak of 615 TF

intel
Xeon E5-2670



166
GFLOPS



M2090



665
GFLOPS

ProLiant SL250 G8
(2CPUs, 3GPUs)



2327
GFLOPS
32/18 GB



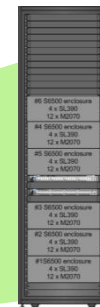
S6500 Chassis
(4 Nodes)



9308
GFLOPS



Rack
(6 Chassis)



55848
GFLOPS



Mellanox 384p FDR Infiniband Switch

Full PCIeG3 X16
bandwidth to all GPUs

Integrated with NICS
Datacenter Lustre and XSEDE

Keeneland System
(11 Compute Racks)

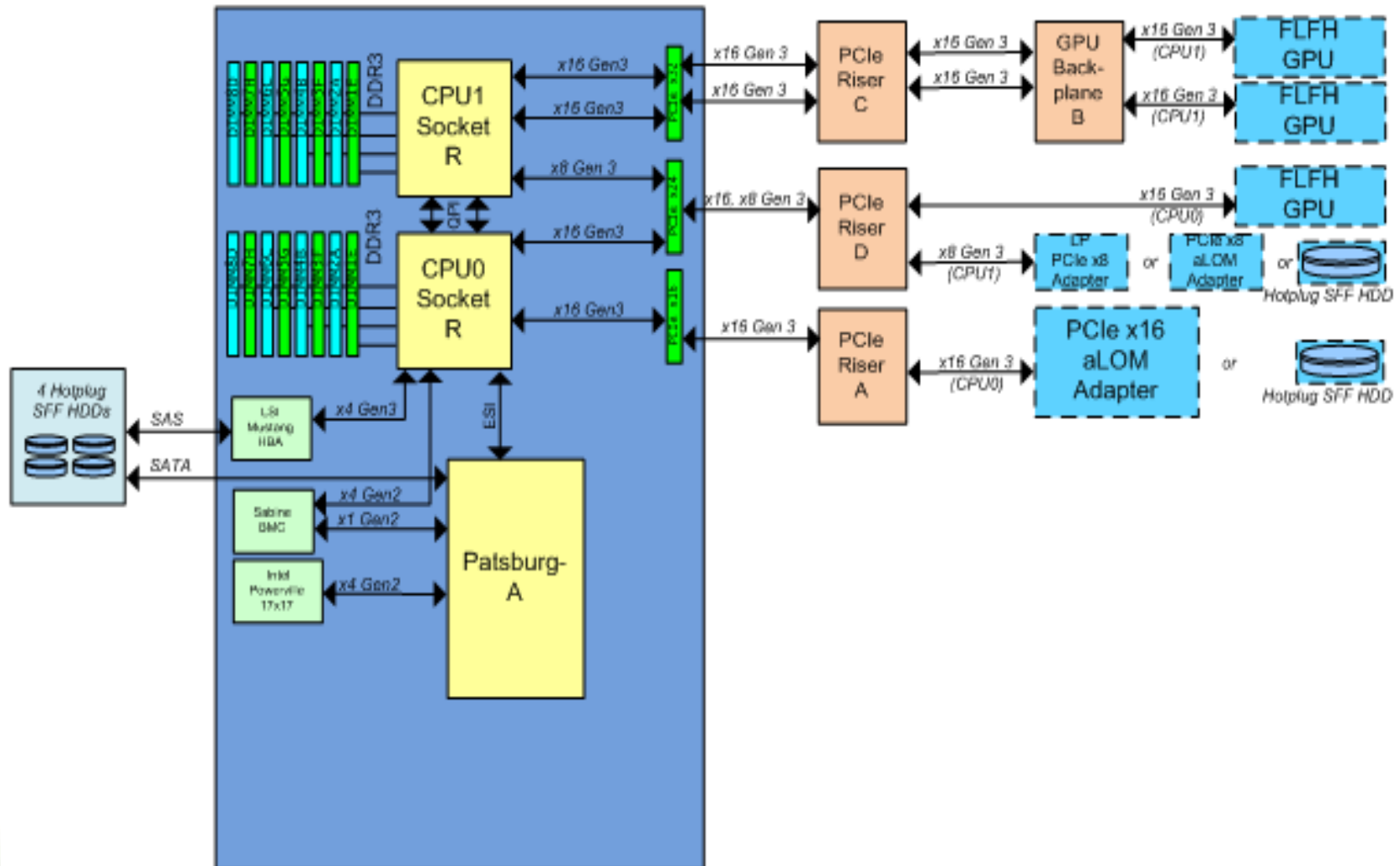


614450
GFLOPS

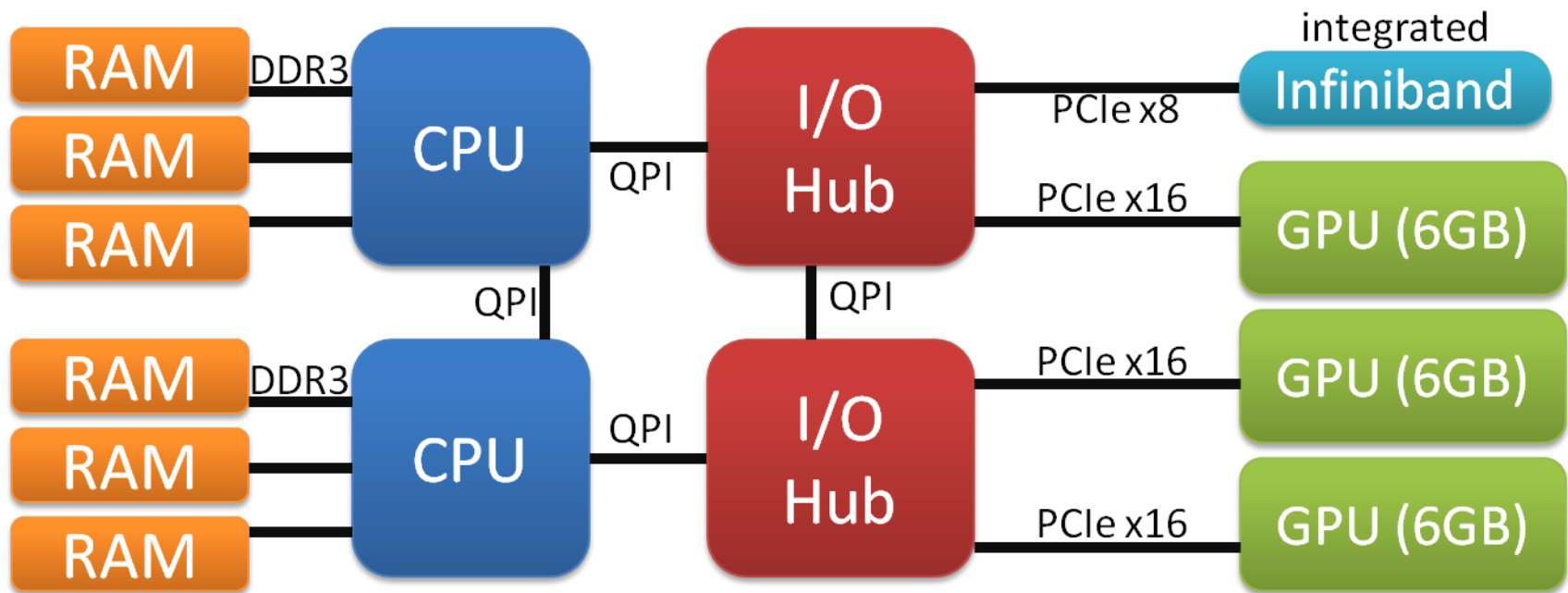
J.S. Vetter, R. Glassbrook et al., "Keeneland: Bringing heterogeneous GPU computing to the computational science community," *IEEE Computing in Science and Engineering*, 13(5):90-5, 2011, <http://dx.doi.org/10.1109/MCSE.2011.83>.



Keeneland Full Scale System Node Architecture

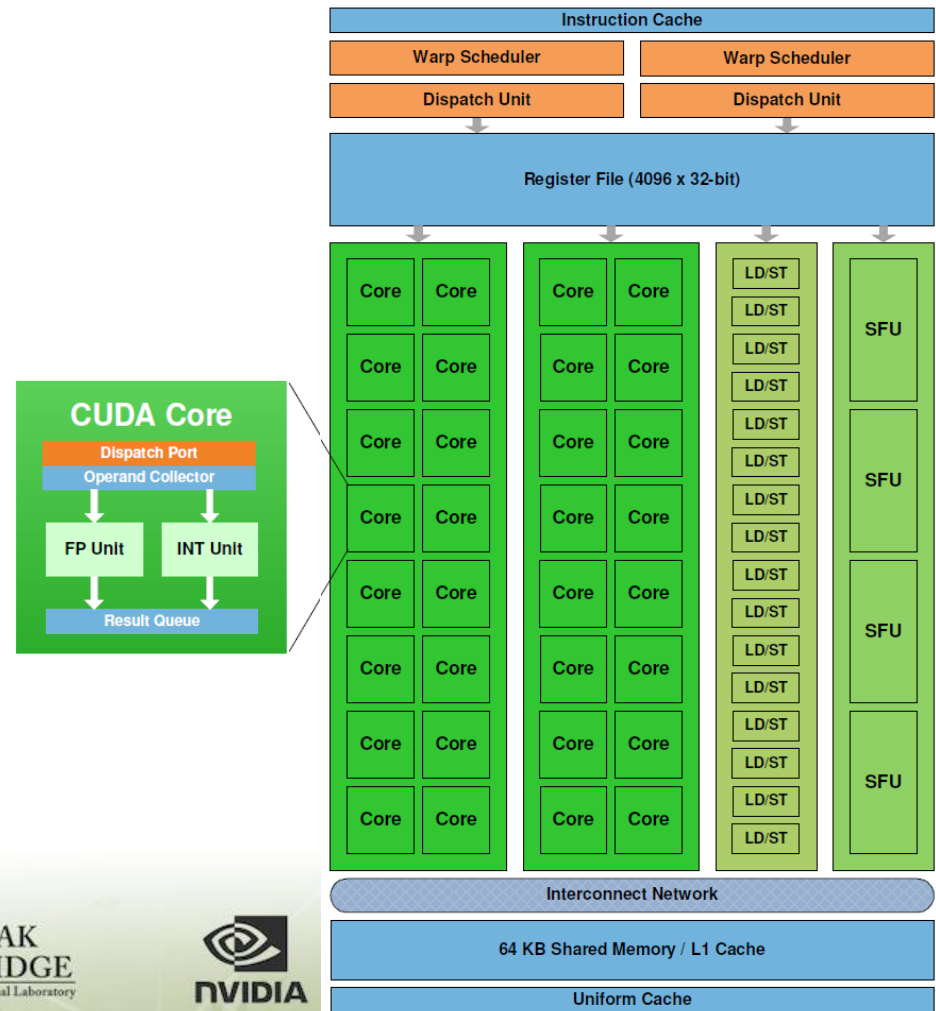


KIDS Node Architecture SL390



NVIDIA Fermi - M2090

- 3B transistors in 40nm
- 512 CUDA Cores
 - New IEEE 754-2008 floating-point standard
 - FMA
 - 8× the peak double precision arithmetic performance over NVIDIA's last generation GPU
 - 32 cores per SM, 21k threads per chip
- 384b GDDR5, 6 GB capacity
 - 178 GB/s memory BW
- C/M2090
 - 665 GigaFLOPS DP, 6GB
 - ECC Register files, L1/L2 caches, shared memory and DRAM

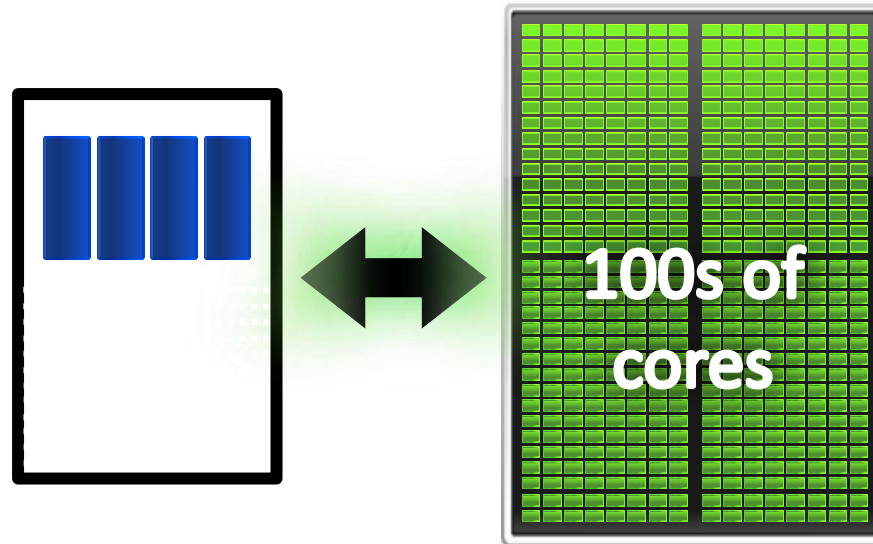


KIDS v. KFS

Item	KID (initial)	KFS (full scale)
Started Operation	Nov 2010 (upgraded April 2012)	October 2012
Node	HP Proliant SL390	HP Proliant SL250
# Nodes	120	264
GPU	M2090 (Fermi) Upgraded from M2070 in Spring 2012	M2090 (Fermi)
# GPUs	360	792
GPU Peak DP	665	665
GPU Mem BW	177	177
GPU DGEMM	470	470
Host PCI	PCIeG2x16	PCIeG3x16
Interconnect	Integrated Mellanox IB QDR	Mellanox IB FDR
IB Ports/node	1	1
IB Switches	Qlogic QDR 384	Mellanox FDR 384p Switch
Memory/node	24	32
Host CPU	Westmere	Sandy Bridge
GPU/CPU Ratio	3:2	3:2
Racks	7	13
DP Peak (GPUs only) (TF)	239	527

Heterogeneous Computing with GPUs

CPU + GPU Co-Processing



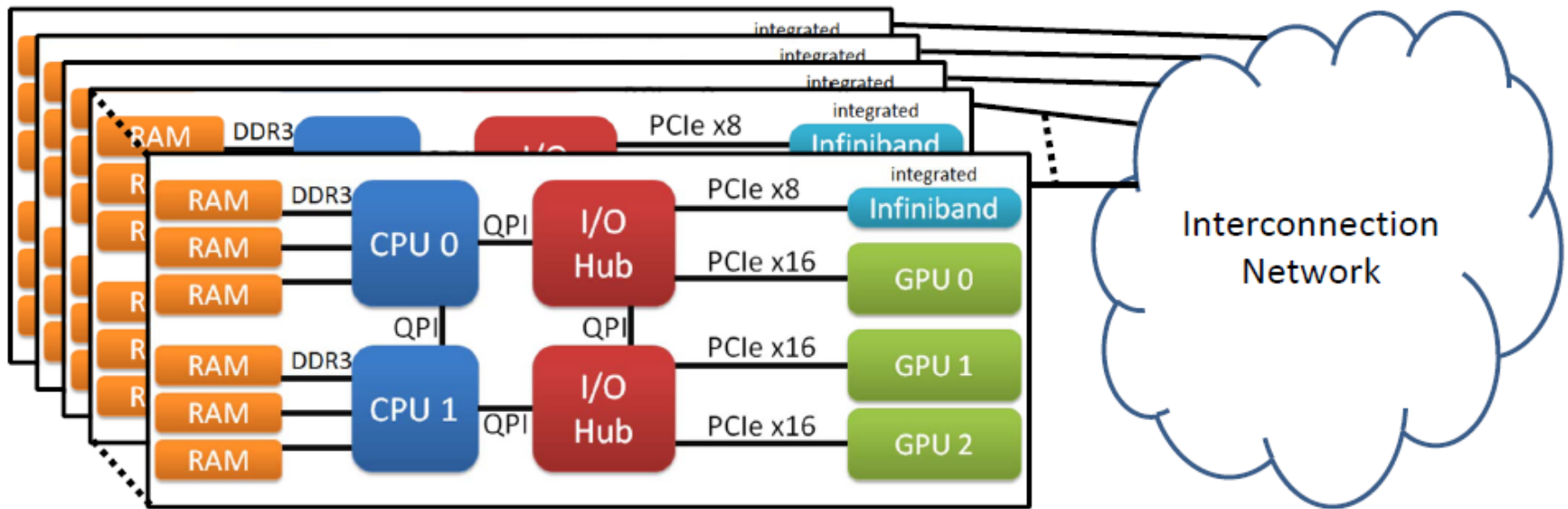
CPU

48 GigaFlops (DP)

GPU

665 GigaFlops (DP)

Applications must use a mix of programming models



MPI

Low overhead

Resource contention

Locality

OpenMP, Pthreads

SIMD

NUMA

OpenACC, CUDA, OpenCL

Memory use, coalescing

Data orchestration

Fine grained parallelism

Hardware features

Keeneland Software Environment

- Integrated with NSF XSEDE
 - Including XSEDE and NICS software stack (cf. Kraken)
- Programming environments
 - CUDA
 - OpenCL
 - Compilers
 - GPU-enabled
 - Scalable debuggers
 - Performance tools
 - Libraries
- Tools and programming options are changing rapidly
 - HMPP, PGI, OpenMPC, R-stream,
- Additional software activities
 - Performance and correctness tools
 - Scientific libraries
 - Virtualization

A Very Brief Introduction to Programming GPUs with CUDA

[nvidia-intro-to-cuda.pdf](#)

Introduction to CUDA C



What is CUDA?

- CUDA Architecture
 - Expose general-purpose GPU computing as first-class capability
 - Retain traditional DirectX/OpenGL graphics performance
- CUDA C
 - Based on industry-standard C
 - A handful of language extensions to allow heterogeneous programs
 - Straightforward APIs to manage devices, memory, etc.
- This talk will introduce you to CUDA C

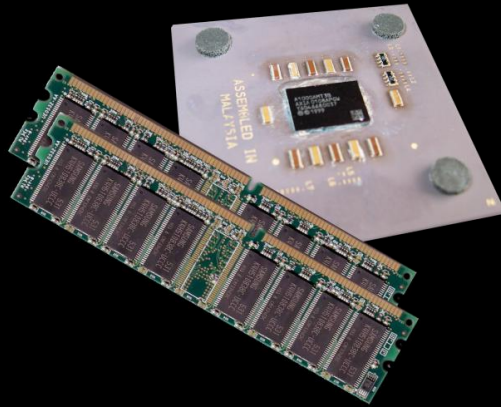
Introduction to CUDA C

- What will you learn today?
 - Start from “Hello, World!”
 - Write and launch CUDA C kernels
 - Manage GPU memory
 - Run parallel kernels in CUDA C
 - Parallel communication and synchronization
 - Race conditions and atomic operations

CUDA C: The Basics

- Terminology
 - **Host** - The CPU and its memory (host memory)
 - **Device** - The GPU and its memory (device memory)

Host



Device



Note: *Figure Not to Scale*

Hello, World!

```
int main( void ) {  
    printf( "Hello, World!\n" );  
    return 0;  
}
```

- This basic program is just standard C that runs on the *host*
- NVIDIA's compiler (`nvcc`) will not complain about CUDA programs with no *device* code
- At its simplest, CUDA C is just C!

Hello, World! with Device Code

```
__global__ void kernel( void ) {  
}
```

```
int main( void ) {  
    kernel<<<1,1>>>();  
    printf( "Hello, World!\n" );  
    return 0;  
}
```

- Two notable additions to the original “Hello, World!”

Hello, World! with Device Code

```
__global__ void kernel( void ) {  
}
```

- CUDA C keyword `__global__` indicates that a function
 - Runs on the device
 - Called from host code
- `nvcc` splits source file into host and device components
 - NVIDIA's compiler handles device functions like `kernel()`
 - Standard host compiler handles host functions like `main()`
 - `gcc`
 - Microsoft Visual C

Hello, World! with Device Code

```
int main( void ) {  
    kernel<<< 1, 1 >>>();  
    printf( "Hello, World!\n" );  
    return 0;  
}
```

- Triple angle brackets mark a call from *host* code to *device* code
 - Sometimes called a “kernel launch”
 - We’ll discuss the parameters inside the angle brackets later
- This is all that’s required to execute a function on the GPU!
- The function `kernel()` does nothing, so this is fairly anticlimactic...

A More Complex Example

- A simple kernel to add two integers:

```
__global__ void add( int *a, int *b, int *c ) {  
    *c = *a + *b;  
}
```

- As before, `__global__` is a CUDA C keyword meaning
 - `add()` will execute on the device
 - `add()` will be called from the host

A More Complex Example

- Notice that we use pointers for our variables:

```
__global__ void add( int *a, int *b, int *c ) {  
    *c = *a + *b;  
}
```

- `add()` runs on the device...so `a`, `b`, and `c` must point to device memory
- How do we allocate memory on the GPU?

Memory Management

- Host and device memory are distinct entities
 - Device pointers point to GPU memory
 - May be passed to and from host code
 - May not be dereferenced from host code
 - Host pointers point to CPU memory
 - May be passed to and from device code
 - May not be dereferenced from device code
- Basic CUDA API for dealing with device memory
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar to their C equivalents, `malloc()`, `free()`, `memcpy()`



A More Complex Example: add()

- Using our `add()` kernel:

```
__global__ void add( int *a, int *b, int *c ) {  
    *c = *a + *b;  
}
```

- Let's take a look at `main()`...

A More Complex Example: `main()`

```
int main( void ) {  
    int a, b, c;                // host copies of a, b, c  
    int *dev_a, *dev_b, *dev_c; // device copies of a, b, c  
    int size = sizeof( int );   // we need space for an integer  
  
    // allocate device copies of a, b, c  
    cudaMalloc( (void**)&dev_a, size );  
    cudaMalloc( (void**)&dev_b, size );  
    cudaMalloc( (void**)&dev_c, size );  
  
    a = 2;  
    b = 7;
```

A More Complex Example: `main()` (cont)

```
// copy inputs to device
cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );

// launch add() kernel on GPU, passing parameters
add<<< 1, 1 >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );

cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Parallel Programming in CUDA C

- But wait...GPU computing is about massive parallelism
- So how do we run code in parallel on the device?
- Solution lies in the parameters between the triple angle brackets:

```
add<<< 1, 1 >>>( dev_a, dev_b, dev_c );
```



```
add<<< N, 1 >>>( dev_a, dev_b, dev_c );
```

- Instead of executing `add()` once, `add()` executed `N` times in parallel

Parallel Programming in CUDA C

- With `add()` running in parallel...let's do vector addition
- Terminology: Each parallel invocation of `add()` referred to as a *block*
- Kernel can refer to its block's index with the variable `blockIdx.x`
- Each block adds a value from `a[]` and `b[]`, storing the result in `c[]`:

```
__global__ void add( int *a, int *b, int *c ) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- By using `blockIdx.x` to index arrays, each block handles different indices

Parallel Programming in CUDA C

- We write this code:

```
__global__ void add( int *a, int *b, int *c ) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- This is what runs in parallel on the device:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

Parallel Addition: add()

- Using our newly parallelized add() kernel:

```
__global__ void add( int *a, int *b, int *c ) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- Let's take a look at main()...

Parallel Addition: main()

```
#define N 512
int main( void ) {
    int *a, *b, *c;           // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
    int size = N * sizeof( int ); // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition: main () (cont)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with N parallel blocks
add<<< N, 1 >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Review

- Difference between “host” and “device”
 - Host = CPU
 - Device = GPU
- Using `__global__` to declare a function as device code
 - Runs on device
 - Called from host
- Passing parameters from host code to a device function

Review (cont)

- Basic device memory management
 - `cudaMalloc()`
 - `cudaMemcpy()`
 - `cudaFree()`
- Launching parallel kernels
 - Launch `N` copies of `add()` with: `add<<< N, 1 >>>();`
 - Used `blockIdx.x` to access block's index

Threads

- Terminology: A block can be split into parallel *threads*
- Let's change vector addition to use parallel threads instead of parallel blocks:

```
__global__ void add( int *a, int *b, int *c ) {  
    c[ threadIdx.x ] = a[ threadIdx.x ] + b[ threadIdx.x ];  
}
```

- We use `threadIdx.x` instead of `blockIdx.x` in `add()`
- `main()` will require one change as well...

Parallel Addition (Threads): main()

```
#define N 512

int main( void ) {
    int *a, *b, *c;           //host copies of a, b, c
    int *dev_a, *dev_b, *dev_c; //device copies of a, b, c
    int size = N * sizeof( int ); //we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition (Threads): main () (cont)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with N threads
add<<< N, N >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

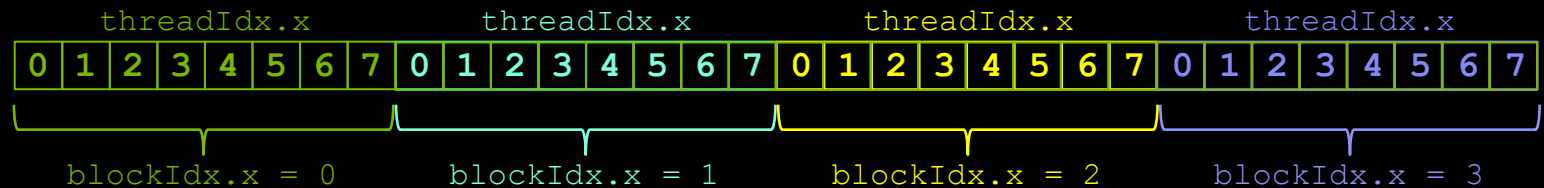
free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Using Threads And Blocks

- We've seen parallel vector addition using
 - Many blocks with 1 thread apiece
 - 1 block with many threads
- Let's adapt vector addition to use lots of *both* blocks and threads
- After using threads and blocks together, we'll talk about *why* threads
- First let's discuss data indexing...

Indexing Arrays With Threads And Blocks

- No longer as simple as just using `threadIdx.x` or `blockIdx.x` as indices
- To index array with 1 thread per entry (using 8 threads/block)



- If we have `M` threads/block, a unique array index for each entry given by

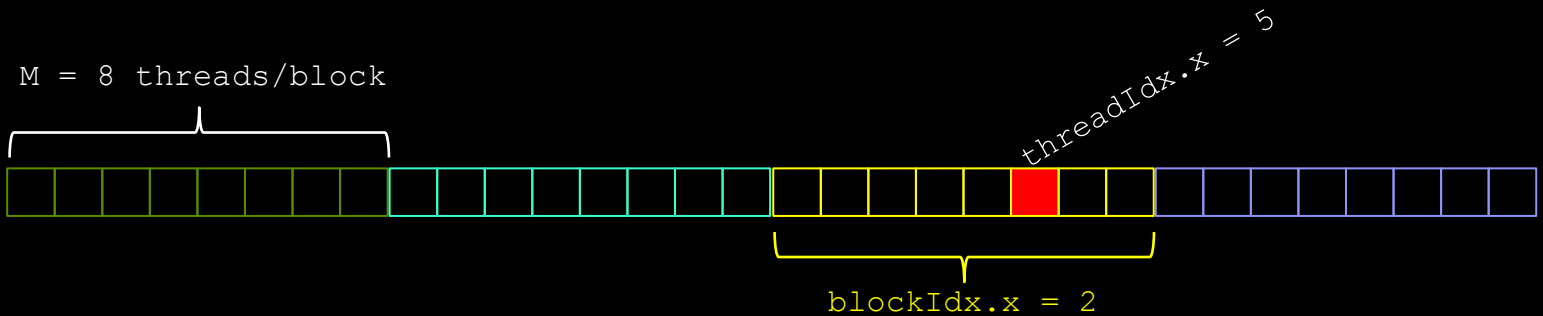
```
int index = threadIdx.x + blockIdx.x * M;
```

↓ ↓ ↓

```
int index =        x                      +        y                      * width;
```

Indexing Arrays: Example

- In this example, the red entry would have an index of 21:



```
int index = threadIdx.x + blockIdx.x * M;  
          =      5      +      2      * 8;  
          = 21;
```

Addition with Threads and Blocks

- The `blockDim.x` is a built-in variable for threads per block:

```
int index= threadIdx.x + blockIdx.x * blockDim.x;
```

- A combined version of our vector addition kernel to use blocks *and* threads:

```
__global__ void add( int *a, int *b, int *c ) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    c[index] = a[index] + b[index];  
}
```

- So what changes in `main()` when we use both blocks and threads?

Parallel Addition (Blocks/Threads): main()

```
#define N    (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;                // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;    // device copies of a, b, c
    int size = N * sizeof( int );  // we need space for N integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```


Parallel Addition (Blocks/Threads): main()

```
// copy inputs to device
```

```
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
```

```
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );
```

```
// launch add() kernel with blocks and threads
```

```
add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c );
```

```
// copy device result back to host copy of c
```

```
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );
```

```
free( a ); free( b ); free( c );
```

```
cudaFree( dev_a );
```

```
cudaFree( dev_b );
```

```
cudaFree( dev_c );
```

```
return 0;
```

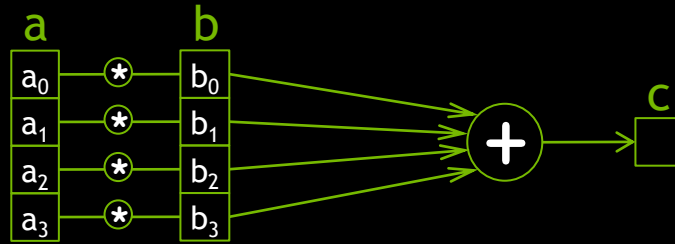
```
}
```

Why Bother With Threads?

- Threads seem unnecessary
 - Added a level of abstraction and complexity
 - What did we gain?
- Unlike parallel blocks, parallel threads have mechanisms to
 - Communicate
 - Synchronize
- Let's see how...

Dot Product

- Unlike vector addition, dot product is a *reduction* from vectors to a scalar



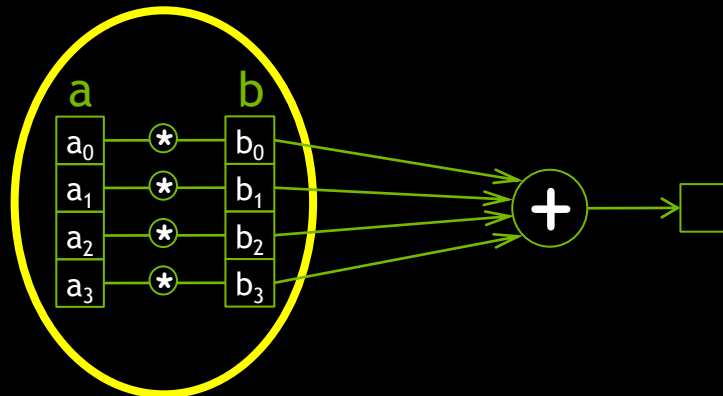
$$c = \vec{a} \cdot \vec{b}$$

$$= (a_0, a_1, a_2, a_3) \cdot (b_0, b_1, b_2, b_3)$$

$$= a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3$$

Dot Product

- Parallel threads have no problem computing the pairwise products:

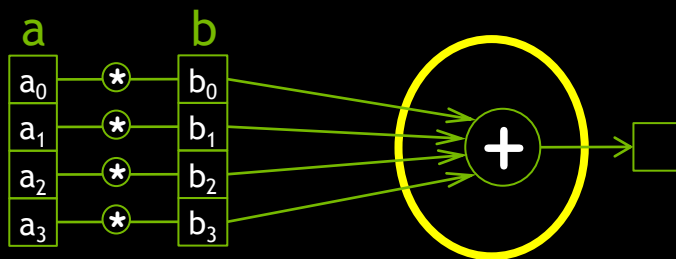


- So we can start a dot product CUDA kernel by doing just that:

```
__global__ void dot( int *a, int *b, int *c )    {  
    // Each thread computes a pairwise product  
    int temp = a[threadIdx.x] * b[threadIdx.x];
```

Dot Product

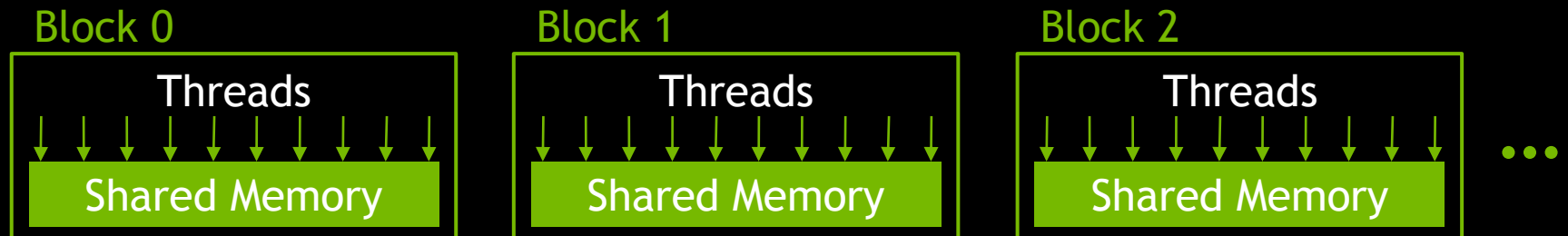
- But we need to share data between threads to compute the final sum:



```
__global__ void dot( int *a, int *b, int *c )    {  
    // Each thread computes a pairwise product  
    int temp = a[threadIdx.x] * b[threadIdx.x];  
  
    // Can't compute the final sum  
    // Each thread's copy of 'temp' is private  
}
```

Sharing Data Between Threads

- Terminology: A block of threads shares memory called...*shared memory*
- Extremely fast, on-chip memory (user-managed cache)
- Declared with the `__shared__` CUDA keyword
- Not visible to threads in other blocks running in parallel



Parallel Dot Product: dot()

- We perform parallel multiplication, serial addition:

```
#define N 512
__global__ void dot( int *a, int *b, int *c ) {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

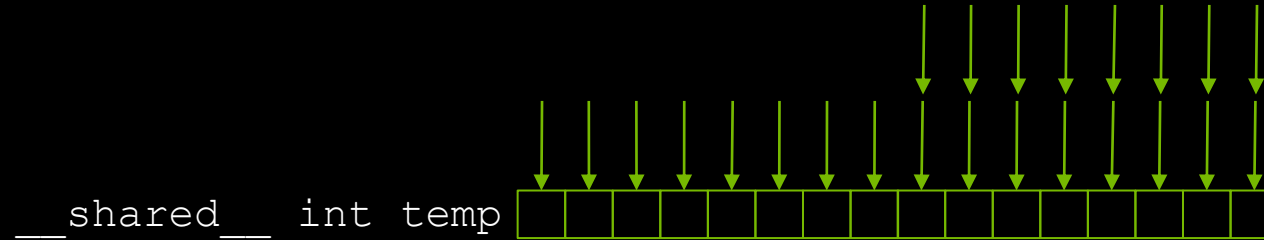
    // Thread 0 sums the pairwise products
    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
            sum += temp[i];
        *c = sum;
    }
}
```


Parallel Dot Product Recap

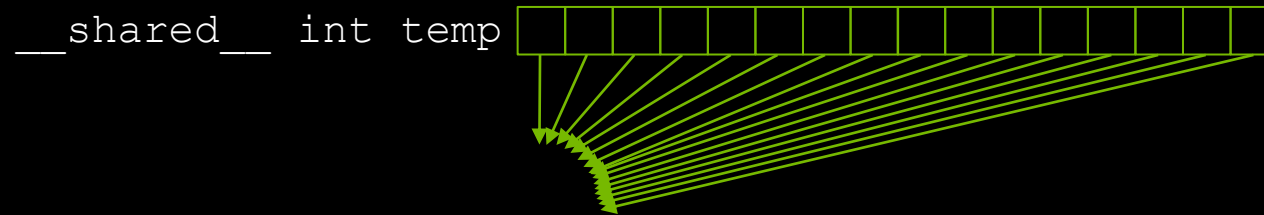
- We perform parallel, pairwise multiplications
- Shared memory stores each thread's result
- We sum these pairwise products from a single thread
- Sounds good...but we've made a huge mistake

Faulty Dot Product Exposed!

- Step 1: In parallel, each thread writes a pairwise product



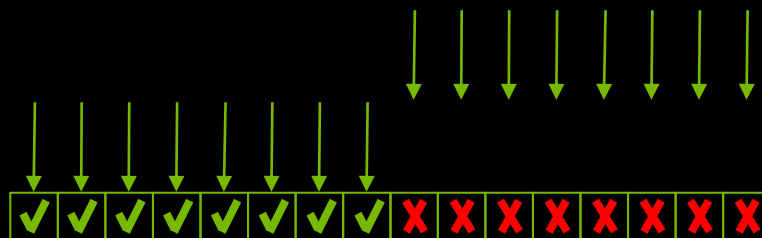
- Step 2: Thread 0 reads and sums the products



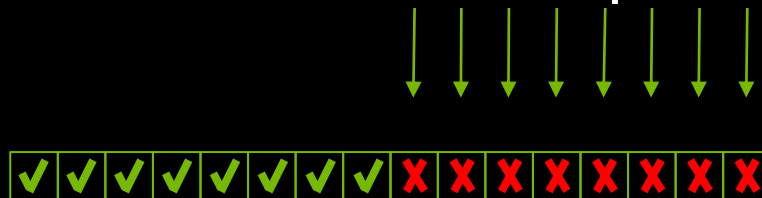
- But there's an assumption hidden in Step 1...

Read-Before-Write Hazard

- Suppose thread 0 finishes its write in step 1



- Then thread 0 reads index 12 in step 2



← This read returns garbage!

- Before thread 12 writes to index 12 in step 1?



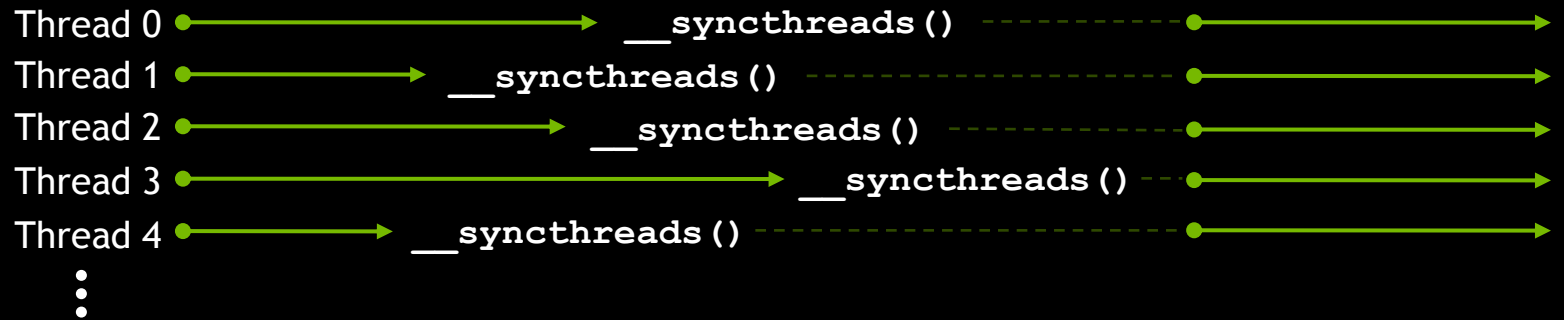
Synchronization

- We need threads to wait between the sections of `dot()`:

```
__global__ void dot( int *a, int *b, int *c ) {  
    __shared__ int temp[N];  
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];  
  
    // * NEED THREADS TO SYNCHRONIZE HERE *  
    // No thread can advance until all threads  
    // have reached this point in the code  
  
    // Thread 0 sums the pairwise products  
    if( 0 == threadIdx.x ) {  
        int sum = 0;  
        for( int i = 0; i < N; i++ )  
            sum += temp[i];  
        *c = sum;  
    }  
}
```

`__syncthreads ()`

- We can synchronize threads with the function `__syncthreads ()`
- Threads in the block wait until *all* threads have hit the `__syncthreads ()`



- Threads are *only* synchronized within a block

Parallel Dot Product: dot ()

```
__global__ void dot( int *a, int *b, int *c ) {  
    __shared__ int temp[N];  
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];  
  
    __syncthreads();  
  
    if( 0 == threadIdx.x ) {  
        int sum = 0;  
        for( int i = 0; i < N; i++ )  
            sum += temp[i];  
        *c = sum;  
    }  
}
```

- With a properly synchronized dot () routine, let's look at main ()

Parallel Dot Product: main()

```
#define N 512

int main( void ) {
    int *a, *b, *c;                // copies of a, b, c
    int *dev_a, *dev_b, *dev_c;    // device copies of a, b, c
    int size = N * sizeof( int );  // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof( int ) );

    a = (int *)malloc( size );
    b = (int *)malloc( size );
    c = (int *)malloc( sizeof( int ) );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Dot Product: main()

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch dot() kernel with 1 block and N threads
dot<<< 1, N >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, sizeof( int ) , cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```


Review

- Launching kernels with parallel threads
 - Launch `add()` with `N` threads: `add<<< 1, N >>>()` ;
 - Used `threadIdx.x` to access thread's index
- Using both blocks and threads
 - Used `(threadIdx.x + blockIdx.x * blockDim.x)` to index input/output
 - `N/THREADS_PER_BLOCK` blocks and `THREADS_PER_BLOCK` threads gave us `N` threads total

Review (cont)

- Using `__shared__` to declare memory as shared memory
 - Data shared among threads in a block
 - Not visible to threads in other parallel blocks
- Using `__syncthreads()` as a barrier
 - No thread executes instructions after `__syncthreads()` until all threads have reached the `__syncthreads()`
 - Needs to be used to prevent *data hazards*

Multiblock Dot Product

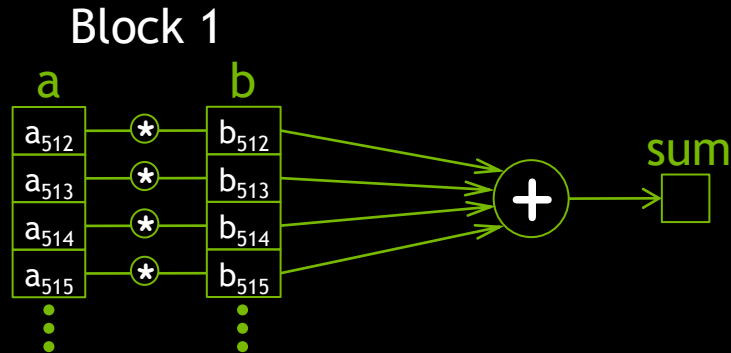
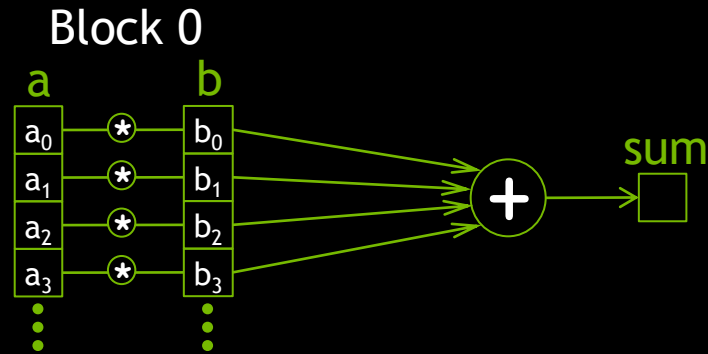
- Recall our dot product launch:

```
// launch dot() kernel with 1 block and N threads  
dot<<< 1, N >>>( dev_a, dev_b, dev_c );
```

- Launching with one block will not utilize much of the GPU
- Let's write a multiblock version of dot product

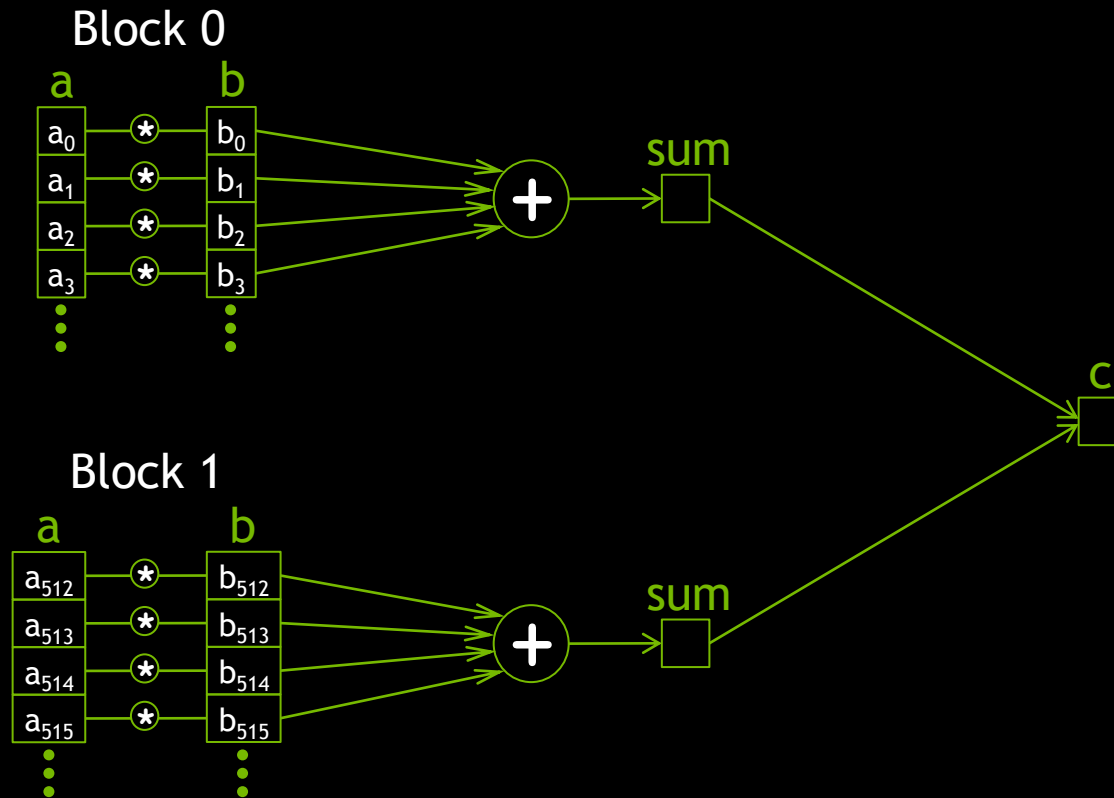
Multiblock Dot Product: Algorithm

- Each block computes a sum of its pairwise products like before:



Multiblock Dot Product: Algorithm

- And then contributes its sum to the final result:



Multiblock Dot Product: dot ()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < THREADS_PER_BLOCK; i++ )
            sum += temp[i];
        atomicAdd( c , sum ); ← was *c += sum;
    }
}
```

- But we have a race condition...
- We can fix it with one of CUDA's atomic operations

Race Conditions

- Terminology: A *race condition* occurs when program behavior depends upon relative timing of two (or more) event sequences
- What actually takes place to execute the line in question: `*c += sum;`
 - Read value at address `c`
 - Add `sum` to value
 - Write result to address `c`

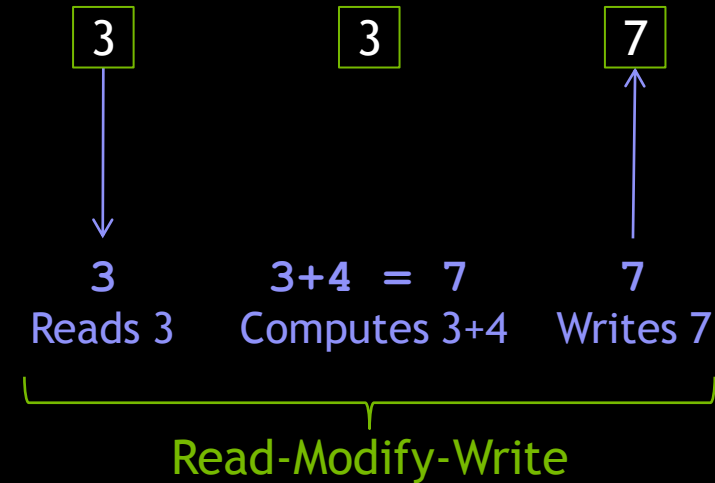
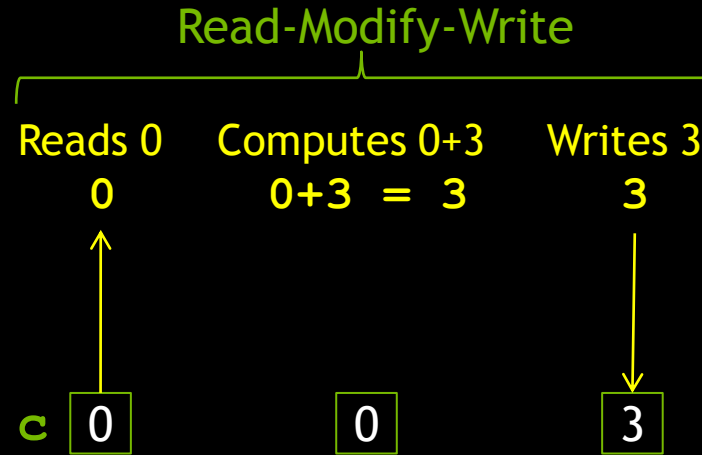
Terminology: *Read-Modify-Write*
- What if two threads are trying to do this at the same time?
 - Thread 0, Block 0
 - Read value at address `c`
 - Add `sum` to value
 - Write result to address `c`
 - Thread 0, Block 1
 - Read value at address `c`
 - Add `sum` to value
 - Write result to address `c`

Global Memory Contention

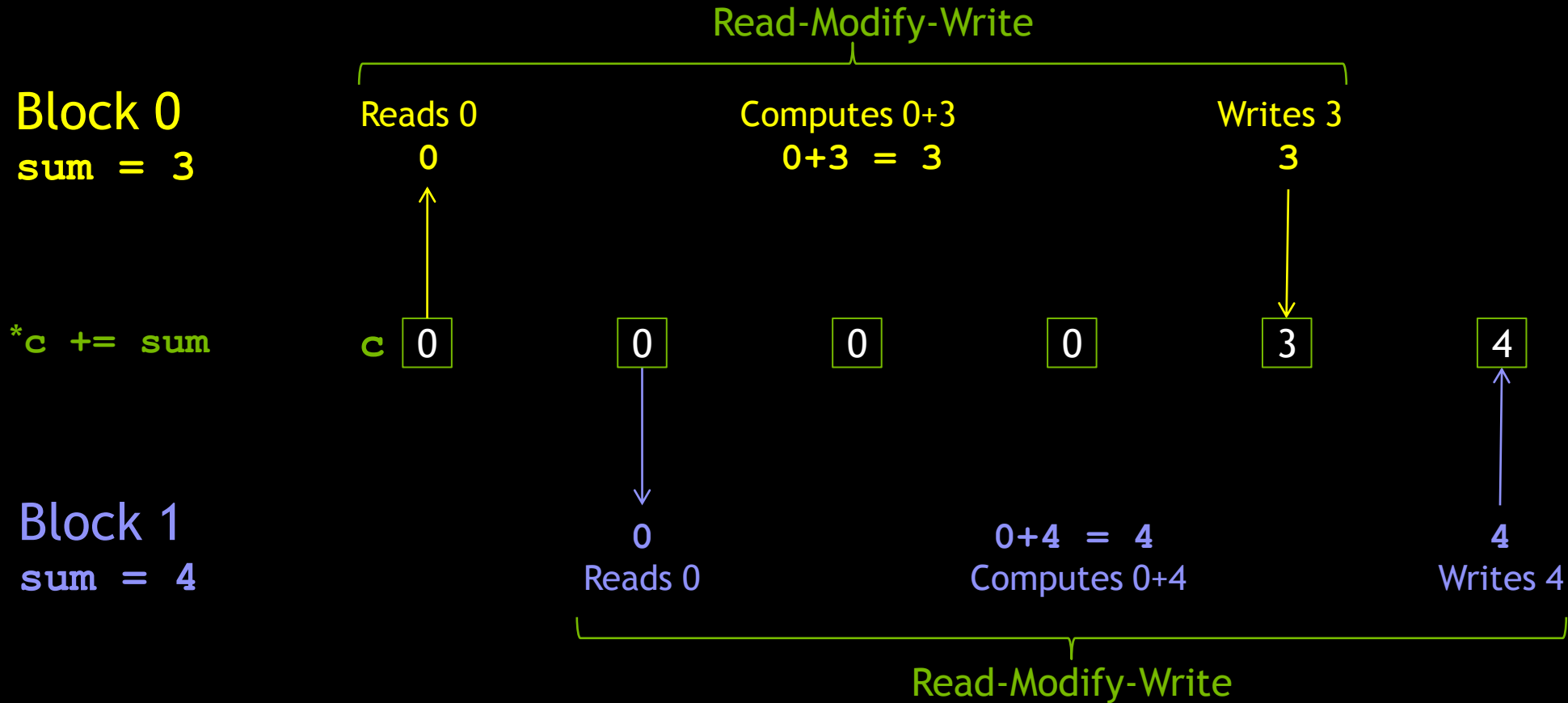
Block 0
sum = 3

*c += sum

Block 1
sum = 4



Global Memory Contention



Atomic Operations

- Terminology: Read-modify-write uninterruptible when *atomic*
- Many *atomic operations* on memory available with CUDA C
 - `atomicAdd()`
 - `atomicSub()`
 - `atomicMin()`
 - `atomicMax()`
 - `atomicInc()`
 - `atomicDec()`
 - `atomicExch()`
 - `atomicCAS()`
- Predictable result when simultaneous access to memory required
- We need to atomically add `sum` to `c` in our multiblock dot product

Multiblock Dot Product: dot ()

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < THREADS_PER_BLOCK; i++ )
            sum += temp[i];
        atomicAdd( c , sum );
    }
}
```

- Now let's fix up `main()` to handle a multiblock dot product

Parallel Dot Product: main()

```
#define N    (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;                // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;    // device copies of a, b, c
    int size = N * sizeof( int );  // we need space for N ints

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof( int ) );

    a = (int *)malloc( size );
    b = (int *)malloc( size );
    c = (int *)malloc( sizeof( int ) );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Dot Product: main()

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch dot() kernel
dot<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, sizeof( int ) , cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Review

- Race conditions
 - Behavior depends upon relative timing of multiple event sequences
 - Can occur when an implied read-modify-write is interruptible
- Atomic operations
 - CUDA provides read-modify-write operations guaranteed to be atomic
 - Atomics ensure correct results when multiple threads modify memory

N-Body

N-Body Algorithms

- An N-body simulation numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body
 - Galaxies
 - Protein folding
 - Molecular dynamics, Materials Science
 - Fluid flow
 - Global illumination (for CG)
- Algorithms
 - All-pairs interactions
 - Computationally intense
 - $O(N^2)$
 - Easily parallelized
 - Usually use some sort of cutoff radius and an approximation for long range forces
- Extensively studied for decades
 - Barnes-Hut, FMM, Particle-mesh

BASIC ALL-PAIRS N-BODY

Example from H. Nguyen, *GPU Gems 3: Addison-Wesley Professional*, 2007.

Basic All-Pairs N-Body

- Each body has
 - Position (x, y, z)
 - Velocity (x, y, z)
 - Mass
 - Perhaps other attributes based on specific simulation

$$\mathbf{f}_{ij} = G \frac{m_i m_j}{\|\mathbf{r}_{ij}\|^2} \cdot \frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|},$$

$$\mathbf{F}_i = \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \mathbf{f}_{ij} = G m_i \cdot \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^3}.$$

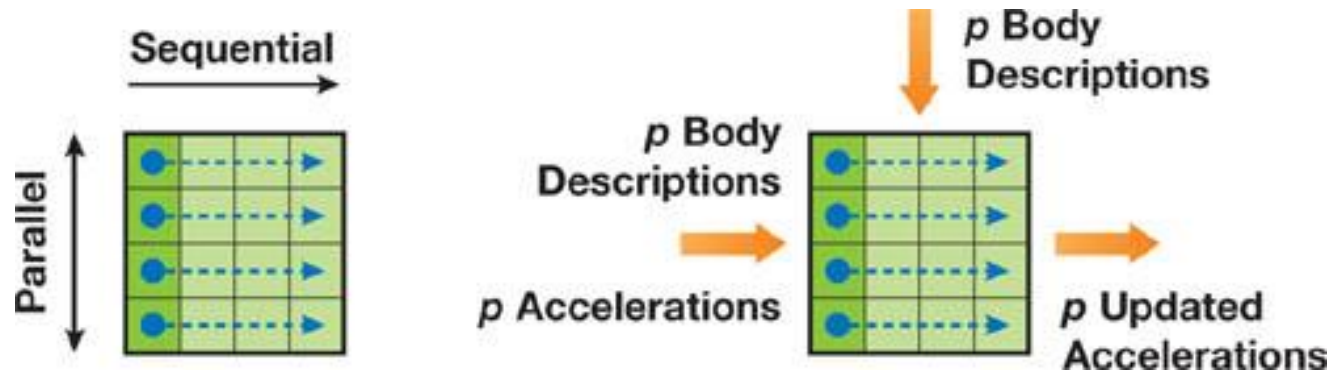
$$\mathbf{F}_i \approx G m_i \cdot \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r}_{ij}}{\left(\|\mathbf{r}_{ij}\|^2 + \epsilon^2\right)^{3/2}}.$$

Implementation Strategy

- Think of the all-pairs algorithm as calculating each entry \mathbf{f}_{ij} in an $N \times N$ grid of all pair-wise force
- Then, total force \mathbf{F}_i (or acceleration \mathbf{a}_i) on body i is obtained from the sum of all entries in row i (a reduction!)
- Abundant parallelism: $O(N^2)$
- But requires $O(N^2)$ memory and needs BW
- Need to improve data reuse to increase computational intensity

Alternate Strategy: Tiles

- Rather, use a tile, which is a square region of this grid that has p rows and p columns
- Only $2p$ body descriptions are necessary to evaluate tile (p can be optimized to fit into fast memory)
- Each row is evaluated sequentially
- But all p rows can be evaluated in parallel



Body-Body Force Calculation (CUDA)

$$\mathbf{f}_{ij} = G \frac{m_i m_j}{\|\mathbf{r}_{ij}\|^2} \cdot \frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|}, \quad \mathbf{F}_i = \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \mathbf{f}_{ij} = G m_i \cdot \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^3}. \quad \mathbf{F}_i \approx G m_i \cdot \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r}_{ij}}{(\|\mathbf{r}_{ij}\|^2 + \epsilon^2)^{3/2}}.$$

```
01.     __device__ float3
02.     bodyBodyInteraction(float4 bi, float4 bj, float3 ai)
03.     {
04.         float3 r;
05.         // r_ij [3 FLOPS]
06.         r.x = bj.x - bi.x;
07.         r.y = bj.y - bi.y;
08.         r.z = bj.z - bi.z;
09.         // distSqr = dot(r_ij, r_ij) + EPS^2 [6 FLOPS]
10.         float distSqr = r.x * r.x + r.y * r.y + r.z * r.z + EPS2;
11.         // invDistCube = 1/distSqr^(3/2) [4 FLOPS (2 mul, 1 sqrt, 1 inv)]
12.         float distSixth = distSqr * distSqr * distSqr;
13.         float invDistCube = 1.0f/sqrtf(distSixth);
14.         // s = m_j * invDistCube [1 FLOP]
15.         float s = bj.w * invDistCube;
16.         // a_i = a_i + s * r_ij [6 FLOPS]
17.         ai.x += r.x * s;
18.         ai.y += r.y * s;
19.         ai.z += r.z * s;
20.         return ai;
21.     }
```

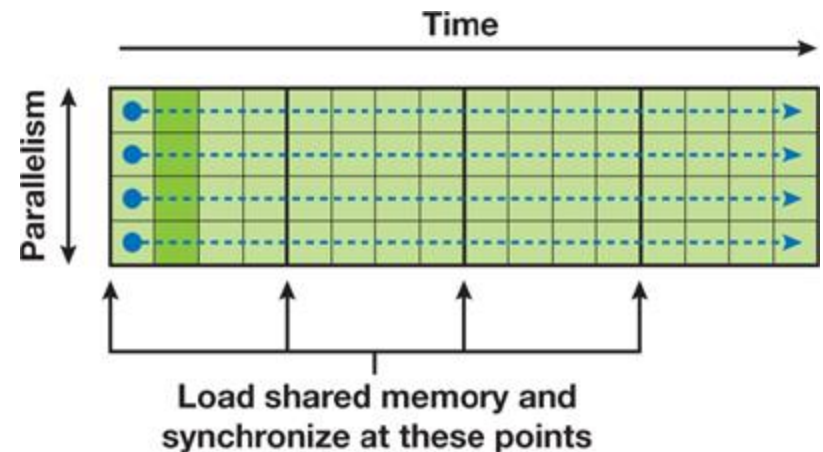
Evaluating a Tile

```
01.     __device__ float3
02.     tile_calculation(float4 myPosition, float3 accel)
03.     {
04.         int i;
05.         extern __shared__ float4[] shPosition;
06.         for (i = 0; i < blockDim.x; i++) {
07.             accel = bodyBodyInteraction(myPosition, shPosition[i], accel);
08.         }
09.         return accel;
10.     }
```

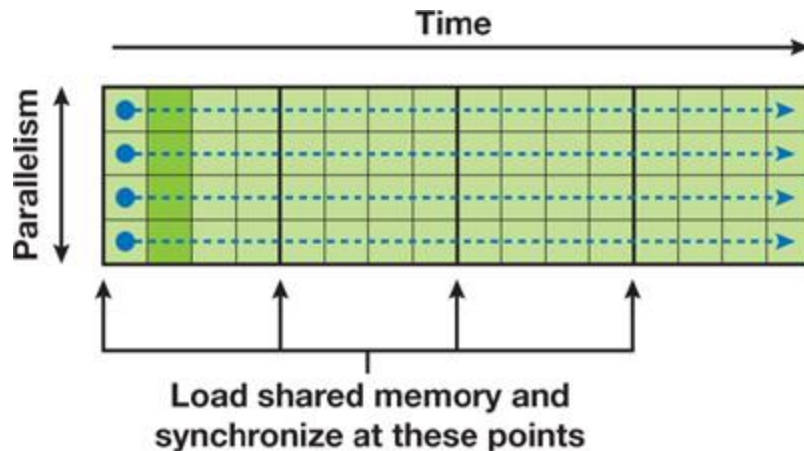
- Each thread will be executing this routine
- shPosition is an array in shared memory

Clustering Tiles into Thread Blocks

- Tiles must be sized to balance parallelism with data reuse
- Parallelism
 - Enough work to keep thread units busy and hide latency
- Reuse
 - Grows w/ number of columns
- Balance
 - Tile size determines register space and shared memory



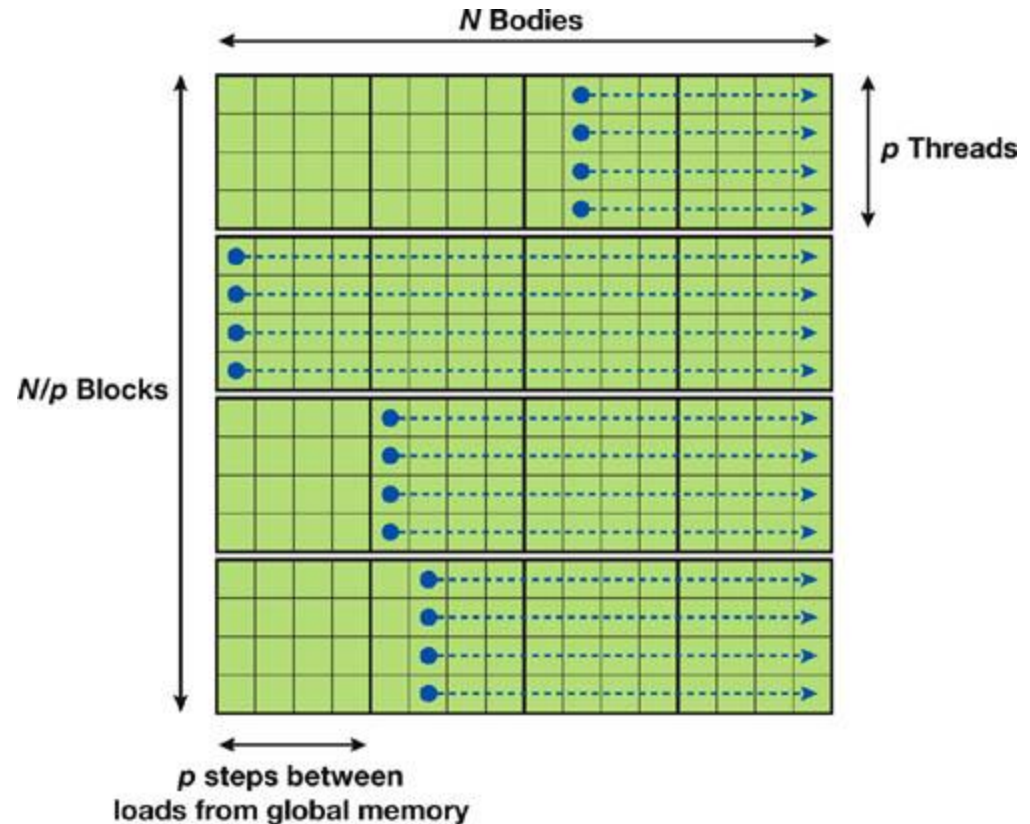
Thread Block Execution



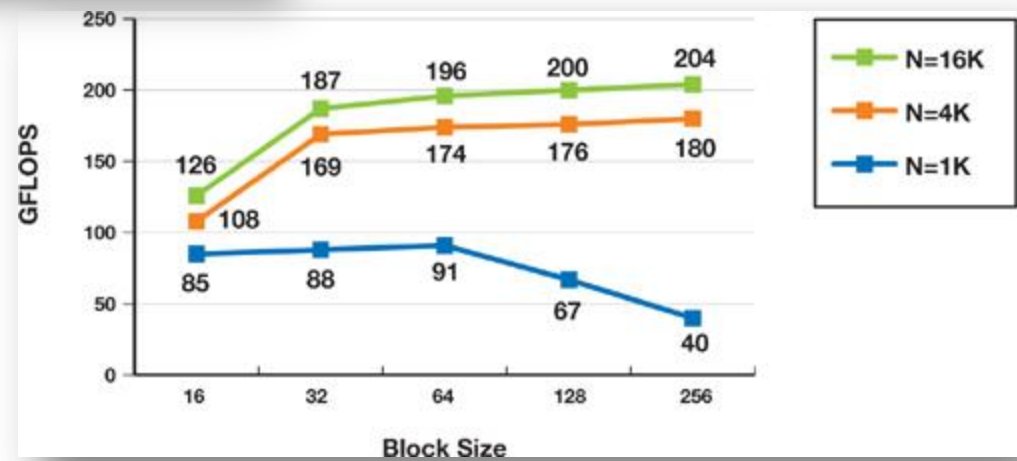
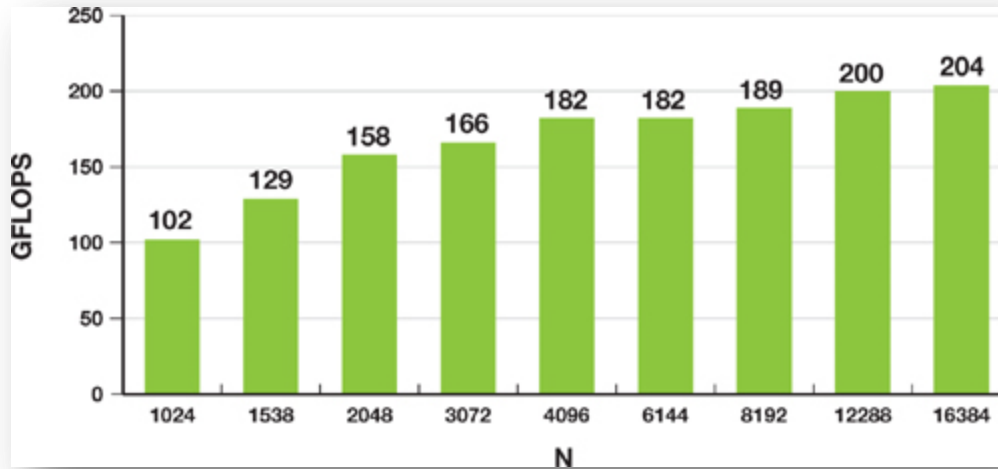
```
01.  __global__ void
02.  calculate_forces(void *devX, void *devA)
03.  {
04.      extern __shared__ float4[] shPosition;
05.      float4 *globalX = (float4 *)devX;
06.      float4 *globalA = (float4 *)devA;
07.      float4 myPosition;
08.      int i, tile;
09.      float3 acc = {0.0f, 0.0f, 0.0f};
10.      int gtid = blockIdx.x * blockDim.x + threadIdx.x;
11.      myPosition = globalX[gtid];
12.      for (i = 0, tile = 0; i < N; i += p, tile++) {
13.          int idx = tile * blockDim.x + threadIdx.x;
14.          shPosition[threadIdx.x] = globalX[idx];
15.          __syncthreads();
16.          acc = tile_calculation(myPosition, acc);
17.          __syncthreads();
18.      }
19.      // Save the result in global memory for the integr
20.      float4 acc4 = {acc.x, acc.y, acc.z, 0.0f};
21.      globalA[gtid] = acc4;
22.  }
```


Grid of Thread Blocks to Calculate All Forces

- 1D grid of N/p independent thread blocks with p threads each



Performance Effects



Caveat: This is the most simple version of n-body

- Barnes-Hut
- Fast Multipole Method
- Particle Mesh, PPPE
- Neutral Territory (Hybrid)
 - Integration step parallelized by assigning particles to processors according to a partitioning of space
 - Force computation step parallelized by pairs across processors but may be unrelated to particle-processor assignments
- A common component of many of these parallel methods for computing long-range forces is the 3-D FFT for solving the Poisson equation on a 3-D mesh

OpenCL (by way of CUDA)

Basic Differences

- terminology
- syntax
- API calls
- compilation

CUDA

- use compiler to build kernels
- C language extensions (nvcc)
 - also a low-level driver-only API
- buffer offsets allowed
- pointer traversal allowed

OpenCL

- build kernels at runtime
- API only; no new compiler
 - API calls to execute kernel
- buffer offsets **not** allowed
- must use pointer arithmetic

Terminology

CUDA	OpenCL
Thread	Work-item
Thread block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

Function Qualifiers

CUDA	OpenCL
__global__	__kernel
__device__	...

Variable Qualifiers

CUDA	OpenCL
__constant__	__constant
__device__	__global
__shared__	__local

Example API Calls

CUDA Version	OpenCL Version
cudaMemcpy	clEnqueueReadBuffer/ clEnqueueWriteBuffer
cudaMalloc	clCreateBuffer
(compile-time call to nvcc)	clBuildProgram
(direct kernel invocation)	clSetKernelArg + clEnqueueNDRangeKernel

Kernel Code Example

CUDA

```
__global__ void
vectorAdd(const float *a,
          const float *b,
          float * c)
{
    // Vector element index
    int nIndex = blockIdx.x *
        blockDim.x + threadIdx.x;

    c[nIndex] = a[nIndex] + b[nIndex];
}
```

OpenCL

```
__kernel void
vectorAdd(__global const float *a,
          __global const float *b,
          __global float * c)
{
    // Vector element index
    int nIndex = get_global_id(0);

    c[nIndex] = a[nIndex] + b[nIndex];
}
```

Host Code Example

CUDA

```
float *data; // device memory allocated with cudaMalloc
int    value;
myfunction<<<nblocks,nthreads>>>(data, value)
```

OpenCL

```
cl_mem data;
int    value;
cl_kernel k = clCreateKernel(prog, "myfunction", 0);
clSetKernelArg(k, 0, sizeof(cl_mem), (void*)&data);
clSetKernelArg(k, 1, sizeof(int),    (void*)&int);
clEnqueueNDRangeKernel(cmdQueue, k, 1, 0, &worksize, 0, 0, 0, 0);
```

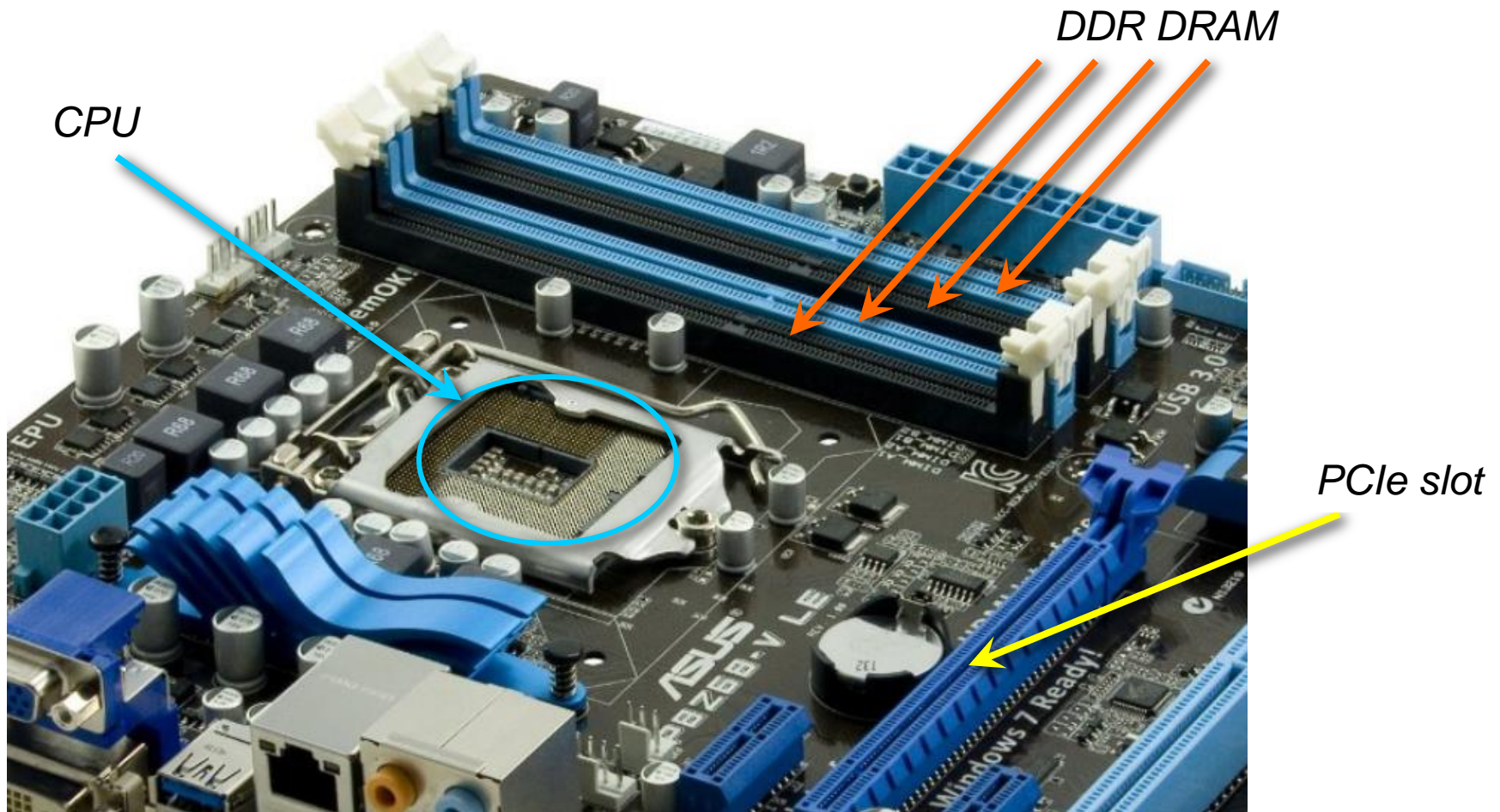
Other Resources

- OpenCL:
 - <http://www.khronos.org/opencv/>
- OpenCL for CUDA programmers:
 - <http://developer.amd.com/zones/opencvzone/programming/pages/portingcudatoopencv.aspx>
 - http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf
- Conversion tools:
 - CU2CL
 - Swan

Advanced Optimization Topics

SINGLE-GPU OPTIMIZATION TECHNIQUES

Host Motherboard Layout



Discrete GPU PCB Layout

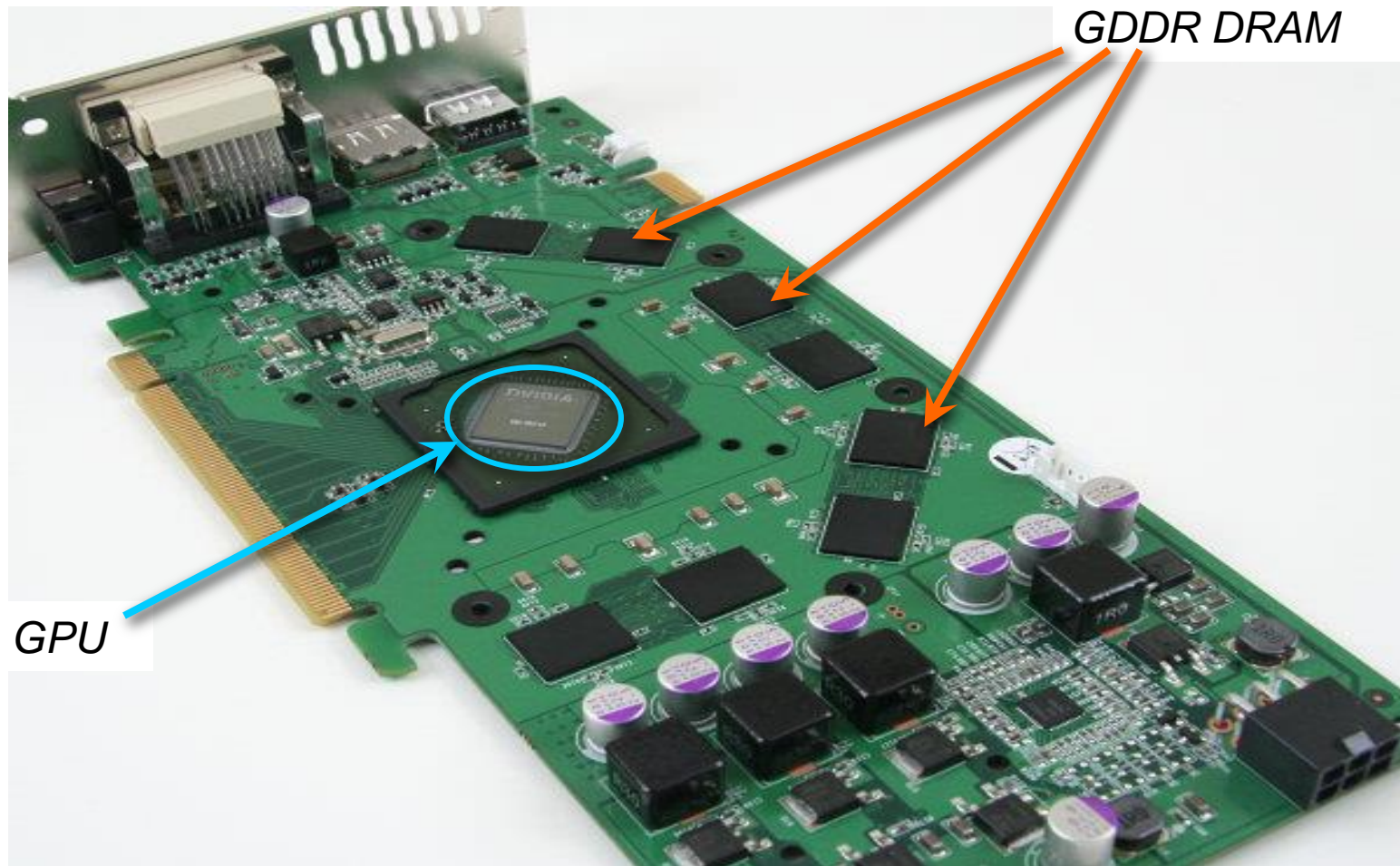
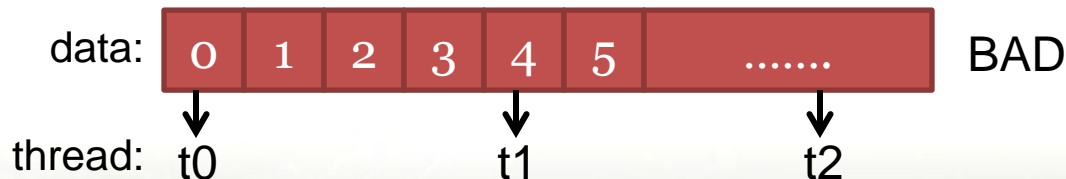
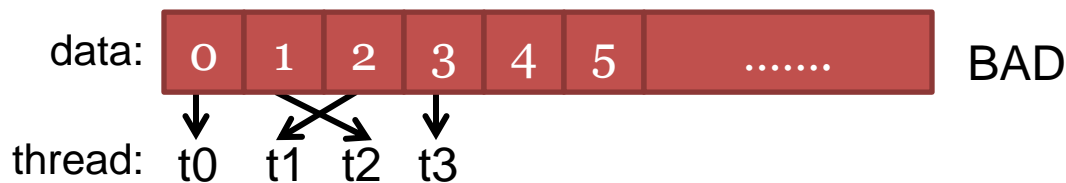


Image from <http://techreport.com/articles.x/14168>

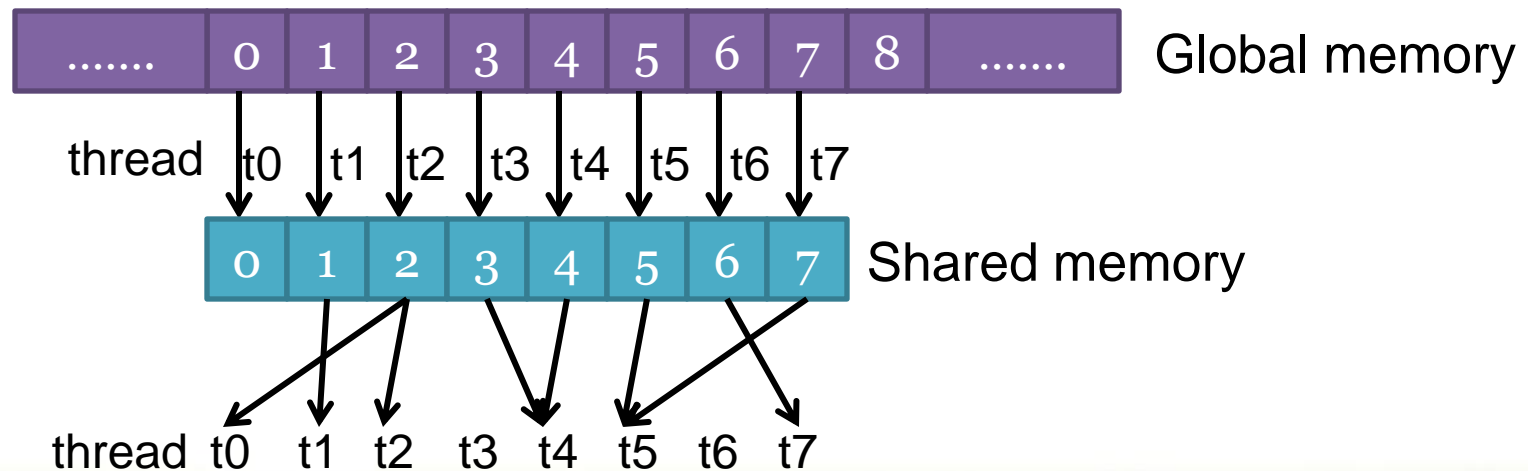
CUDA, OpenCL Optimization

- Minimize data transfers across PCI-Express bus
 - Very expensive: e.g. 5GB/s PCIe *versus* 100GB/s for device
 - Can be asynchronous; overlap communication with computation
- Coalesce memory reads (and writes)
 - ensure threads simultaneously read adjacent values
 - effectively uses GPU memory bandwidth



CUDA, OpenCL Optimization

- Shared memory is fast, local to a group of threads
- When access patterns are irregular:
 - perform coalesced reads to shared memory
 - synchronize threads
 - then access in any pattern



CUDA, OpenCL Optimization

- Unroll loops to minimize overhead
 - GPU kernel compilation not yet mature here
- Execute more than one item per thread
 - further increase computational density
 - remember: maintain coalescing
 - e.g. stride by grid size

*Many presentations, whitepapers detail these aspects of optimization.

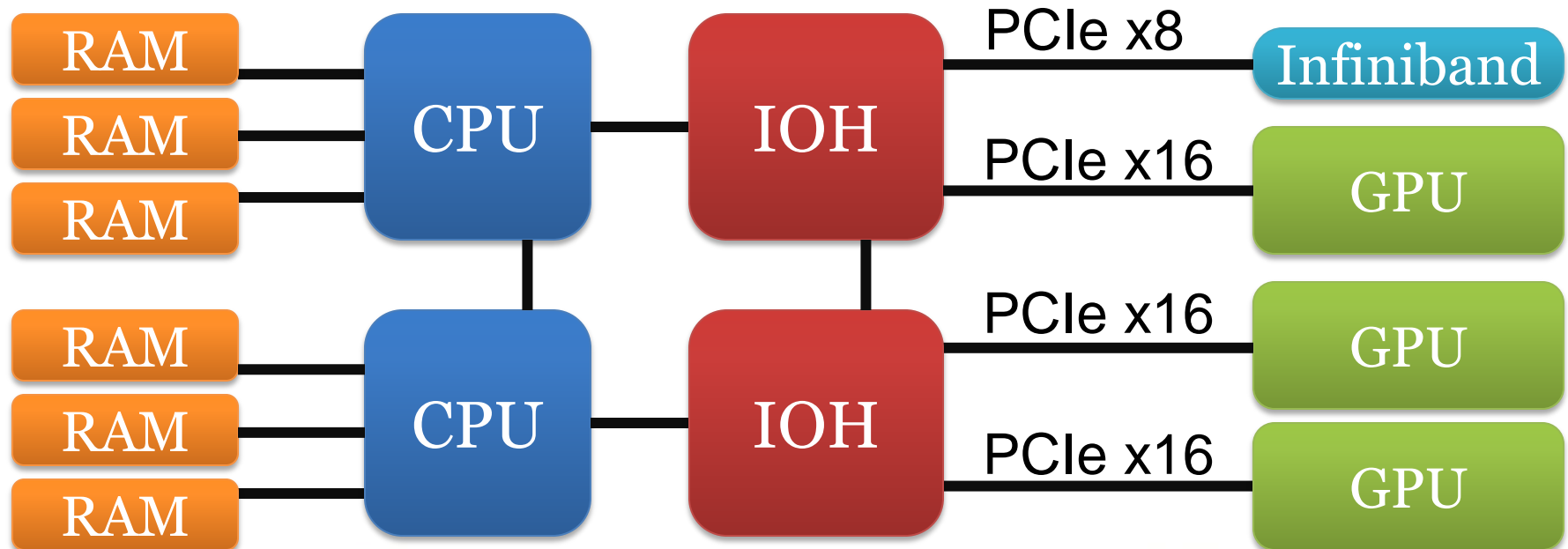
Accelerating Compiler Optimization

- Similar concepts apply
- Relying on compiler for a lot:
 - *coalescing*: you might be able to help by modifying your array layouts
 - *unrolling, tiling, shared memory*: some compilers are better than others, some offer unroll+jam pragmas, some offer shared memory pragmas
 - *minimizing data transfers*: most offer directives to specify allocation and transfer boundaries

OPTIMIZATIONS ON HETEROGENEOUS SYSTEM NODES

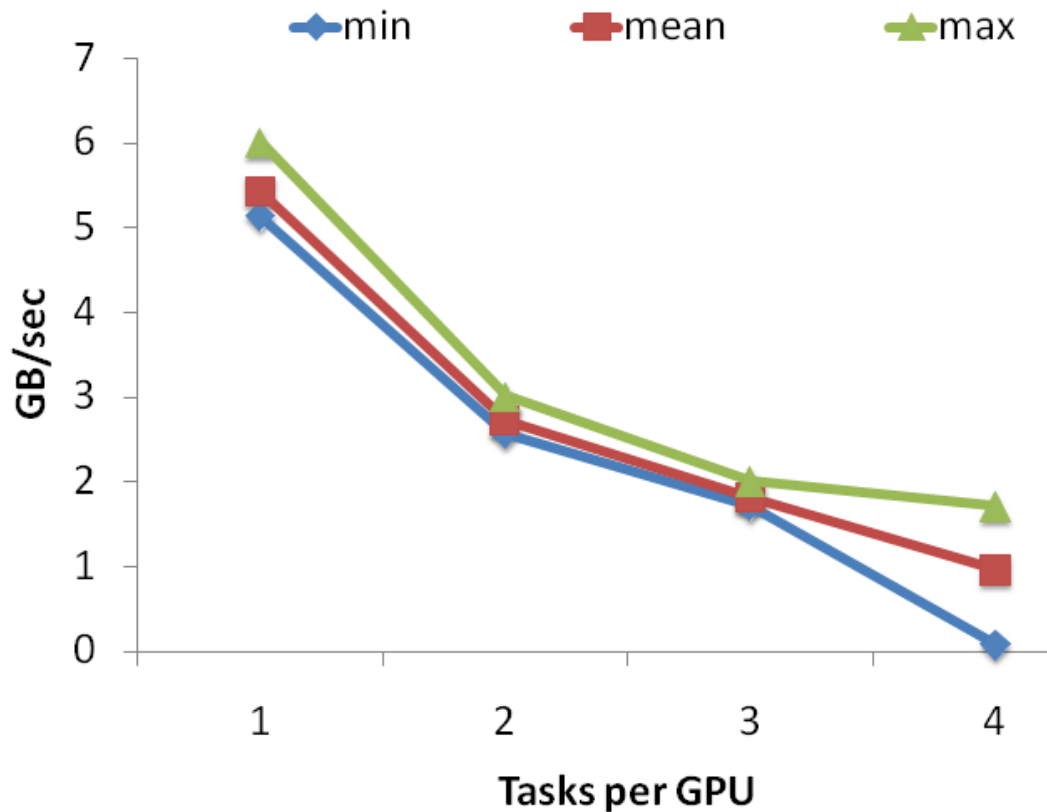
Keeneland's Multi-GPU Nodes

- KIDS is a dual-I/O-hub node architecture
 - Allows full PCIe bandwidth to 3 GPUs and 1 NIC



Sharing GPUs on Keeneland

- Simultaneous PCIe bandwidth to all 3 GPUs



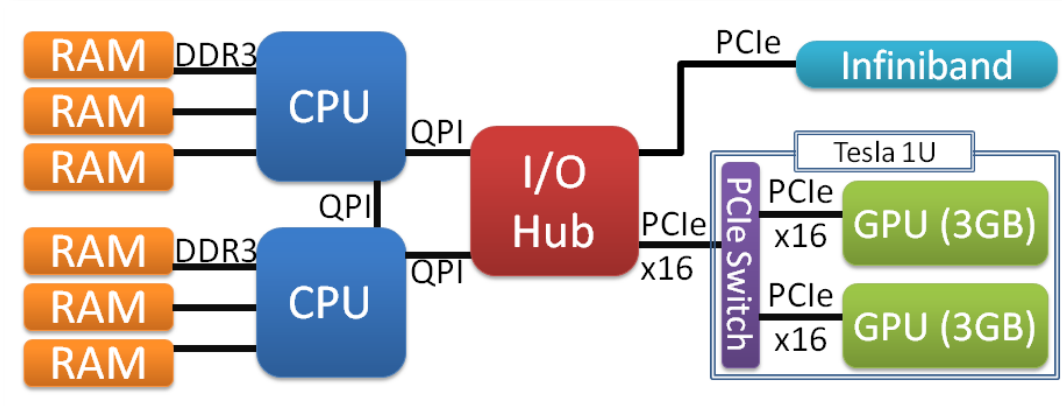
NON-UNIFORM MEMORY ACCESS

Non-Uniform Memory Access

- Node architectures result in Non-Uniform Memory Access (NUMA)
 - Point-to-point connections between devices
 - Not fully-connected topologies
 - Host memory connected to sockets instead of across a bus

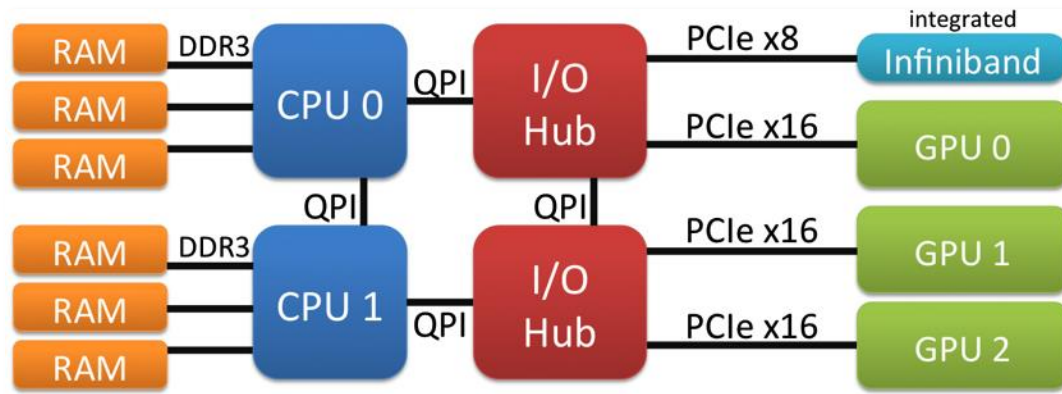
NUMA Can Affect GPUs and Network Too

Older node architecture with single I/O hub but no NUMA effects between CPU and GPU/HCA



- DL160
- Single I/O Hub
- PCIe switch connects GPUs

KIDS node architecture with dual I/O hub but NUMA effects



- SL390
- Dual I/O Hub
- No PCIe switch

NUMA Control Mechanisms

- Process, data placement tools:
 - Tools like libnuma and numactl
 - Some MPI implementations have NUMA controls built in (e.g., Intel MPI, OpenMPI)
- numactl usage:

```
numactl [--interleave=nodes] [--preferred=node]
        [--physcpubind=cpus] [--cpunodebind=nodes]
        [--membind=nodes] [--localalloc] command
numactl [--show]
numactl [--hardware]
```

numactl on KIDS

```
[meredith@kid107]$ numactl -show
```

```
policy: default
```

```
preferred node: current
```

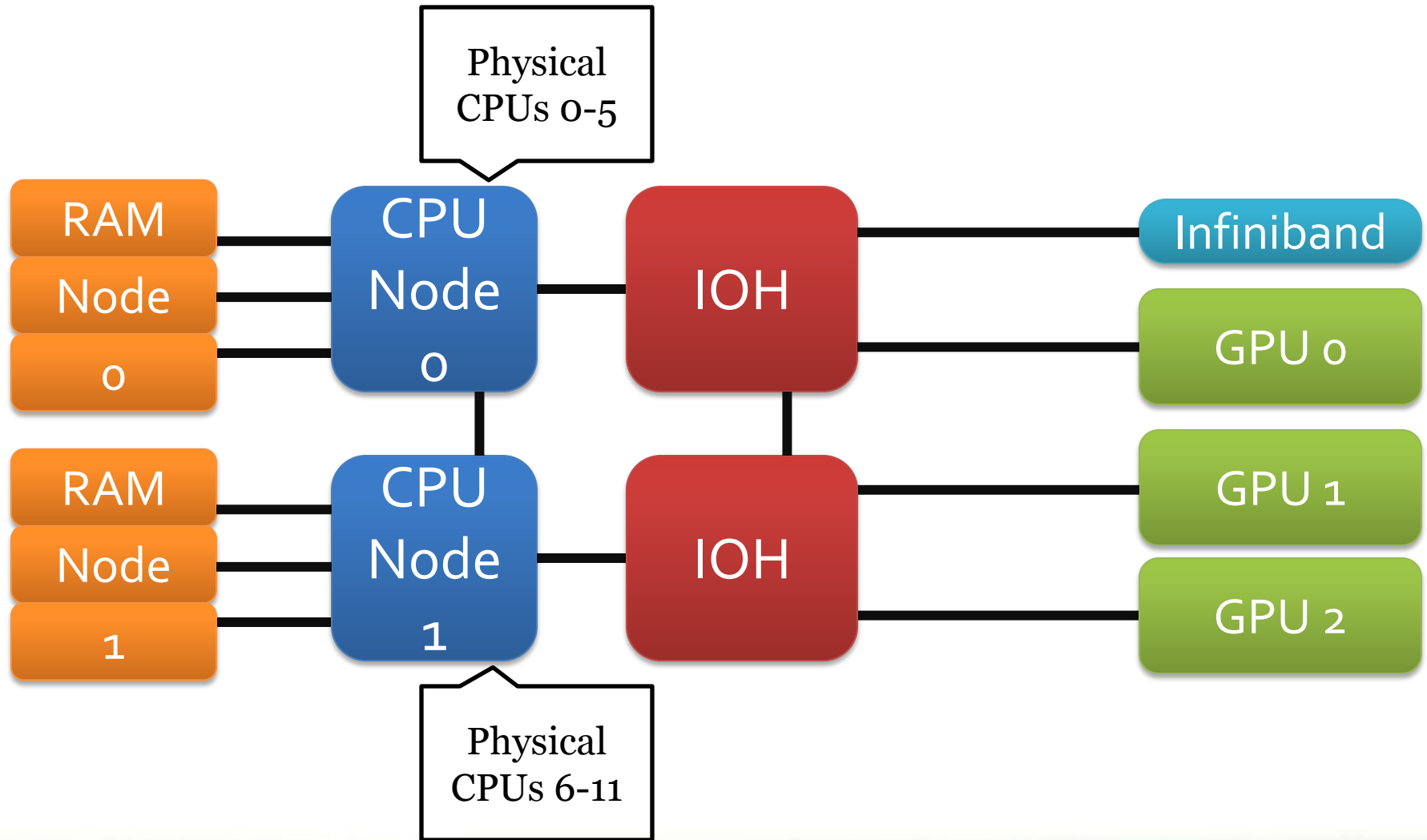
```
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11
```

```
cpubind: 0 1
```

```
nodebind: 0 1
```

```
membind: 0 1
```

“NUMA Nodes” on KIDS nodes



numactl on KIDS

```
[meredith@kid107]$ numactl --hardware
```

```
available: 2 nodes (0-1)
```

```
node 0 size: 12085 MB
```

```
node 0 free: 11286 MB
```

```
node 1 size: 12120 MB
```

```
node 1 free: 11648 MB
```

```
node distances:
```

```
node    0    1
```

```
  0:   10   20
```

```
  1:   20   10
```

OpenMPI with NUMA control

Use *mpirun* to execute a script:

```
mpirun ./prog_with_numa.sh
```

In that script (*prog_with_numa.sh*) launch under *numactl*:

```
if [[ $OMPI_COMM_WORLD_LOCAL_RANK == "0" ]]
then
    numactl --membind=0 --cpunodebind=0 ./prog -args
else
    numactl --membind=1 --cpunodebind=1 ./prog -args
fi
```

How much Does NUMA Impact Performance?

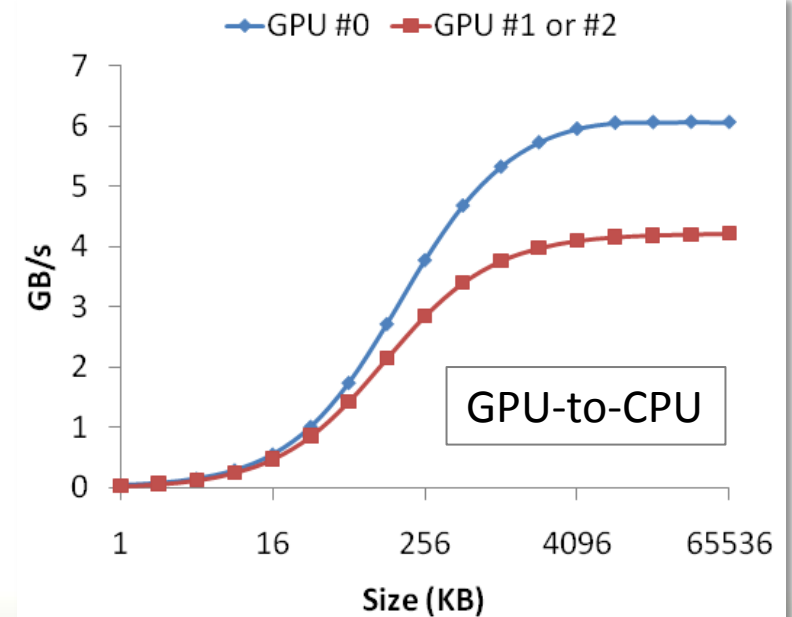
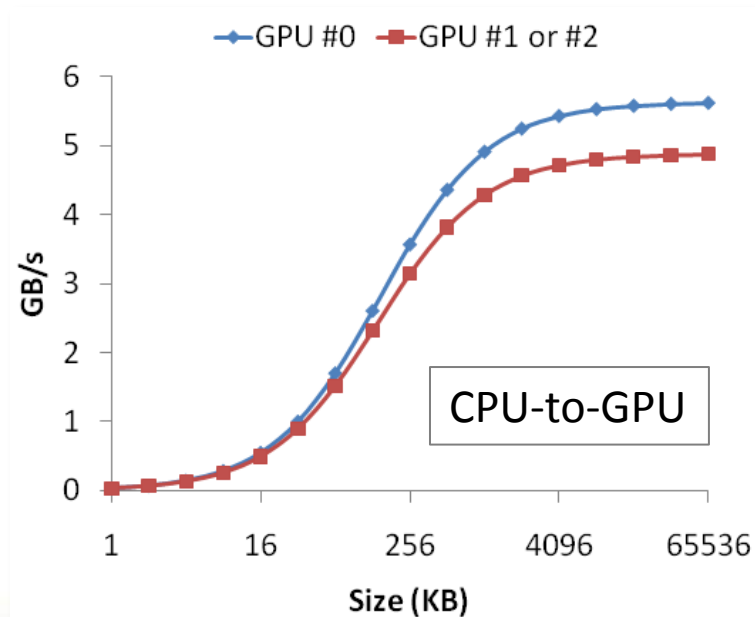
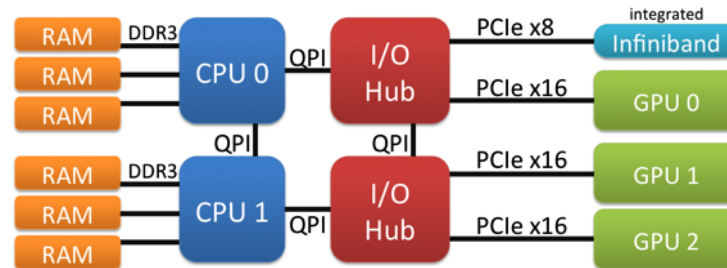
- Microbenchmarks to focus on individual node components
- Macrobenchmarks to focus on individual operations and program kernels
- Full applications to gauge end-user impact

Spafford, K., Meredith, J., Vetter, J. **Quantifying NUMA and Contention Effects in Multi-GPU Systems**. Proceedings of the Fourth Workshop on General-Purpose Computation on Graphics Processors (GPGPU 2011). Newport Beach, CA, USA.

Meredith, J., Roth, P., Spafford, K., Vetter, J. **Performance Implications of Non-Uniform Device Topologies in Scalable Heterogeneous GPU Systems**. IEEE MICRO Special Issue on CPU, GPU, and Hybrid Computing. October 2011.

Data Transfer Bandwidth

- Measured bandwidth of data transfers between CPU socket 0 and the GPUs



SHOC Benchmark Suite

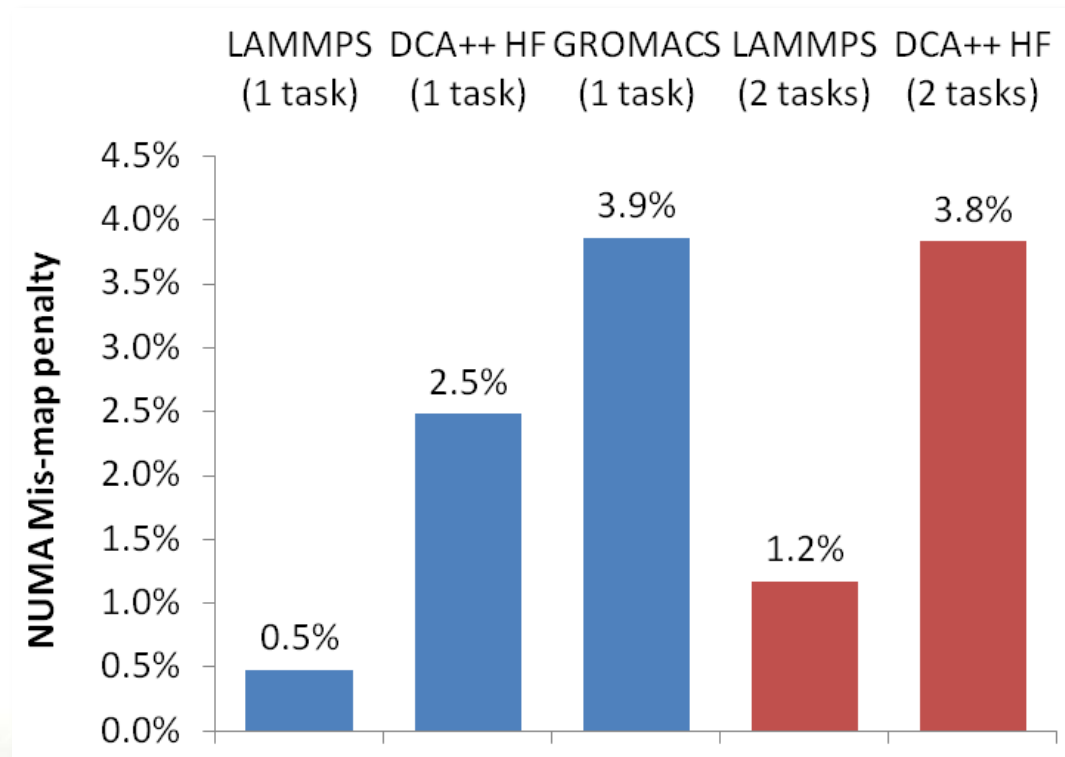
- What penalty for “long” mapping?
- Rough inverse correlation to computational intensity

Test	Units	Correct NUMA	Incorrect NUMA	% Penalty
SGEMM	GFLOPS	535.640	519.581	3%
DGEMM	GFLOPS	239.962	230.809	4%
FFT	GFLOPS	30.501	26.843	12%
FFT-DP	GFLOPS	15.181	13.352	12%
MD	GB/s	12.519	11.450	9%
MD-DP	GB/s	19.063	17.654	7%
Reduction	GB/s	5.631	4.942	12%
Scan	GB/s	0.007	0.005	31%
Sort	GB/s	1.081	0.983	9%
Stencil	seconds	8.749	11.895	36%

Table 3: SHOC Benchmark Results

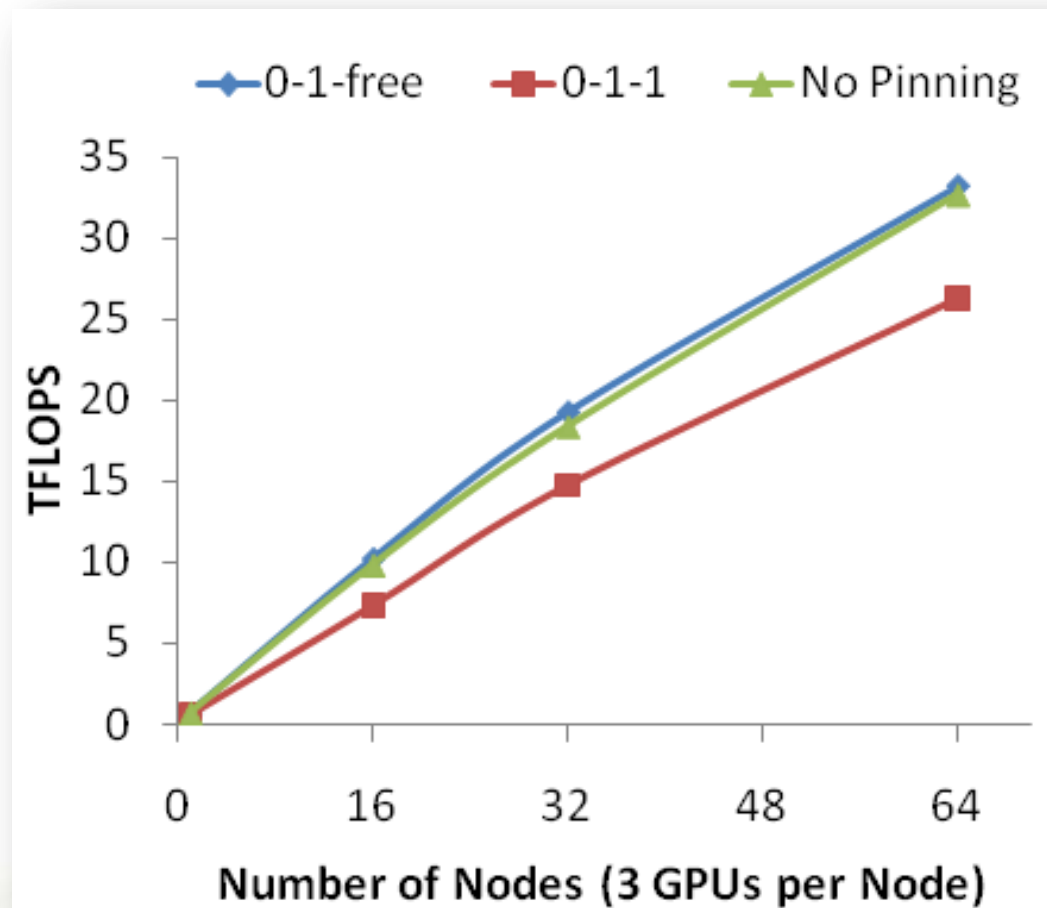
Full Applications

- With one application task, performance penalty for using incorrect mapping (e.g., CPU socket 0 with GPU 1)
- With two application tasks, performance penalty for using mapping that uses “long” paths for both (e.g., CPU socket 0 with GPU 1 and CPU socket 1 with GPU 0)



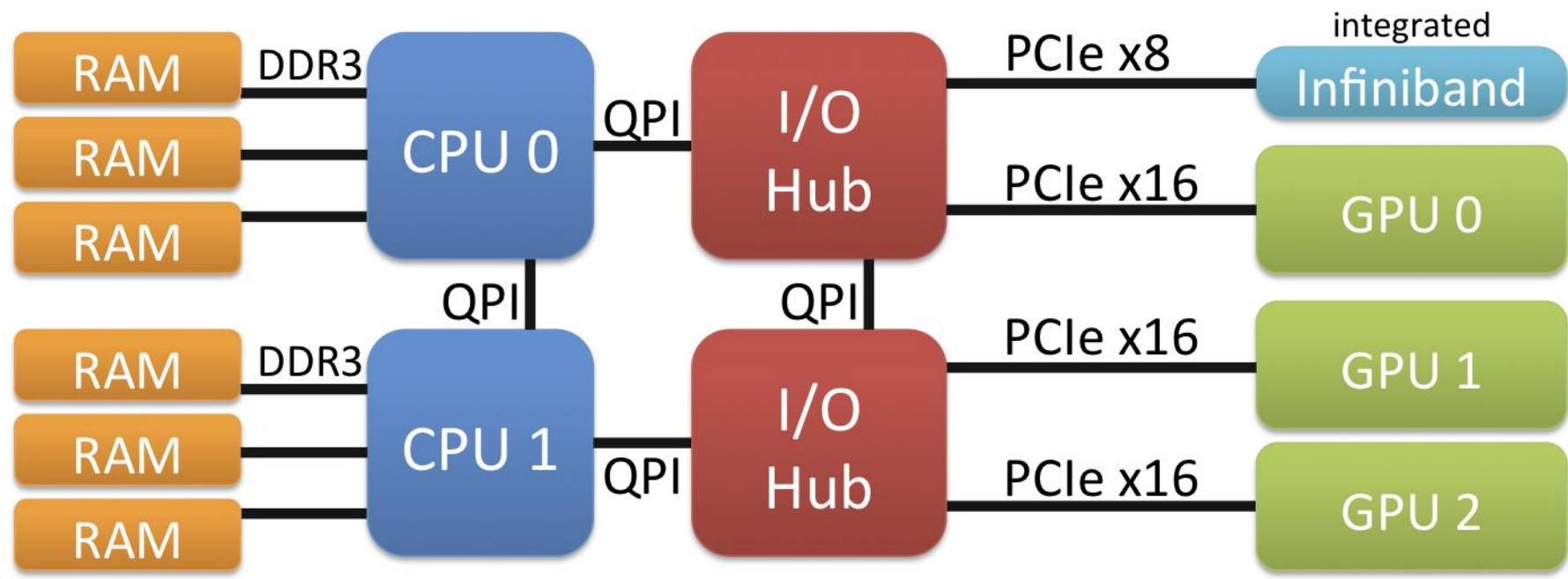
HPL Linpack

- Runtimes on KIDS under 3 pinning scenarios



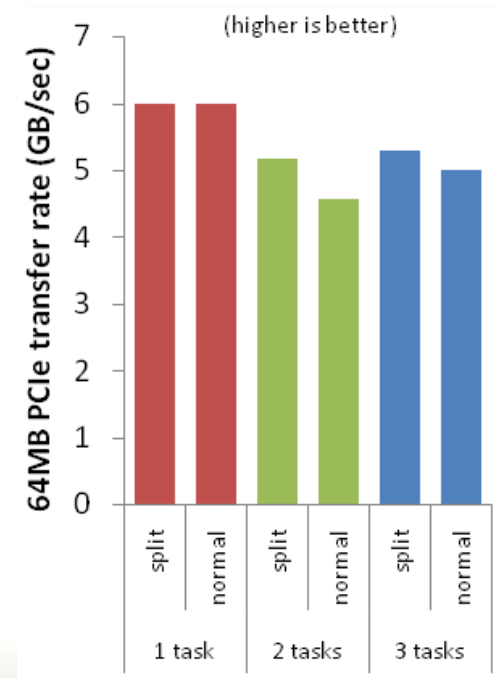
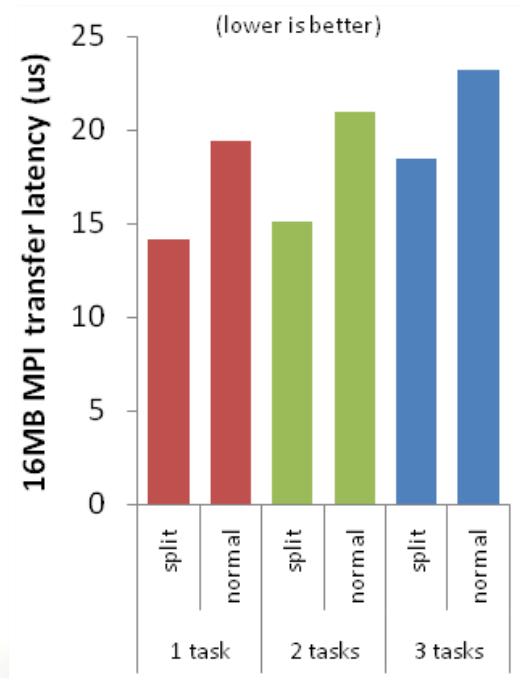
NUMA and Network Traffic

- Have to worry about not only process/data placement for CPU and GPU, but also about CPU and Infiniband HCA

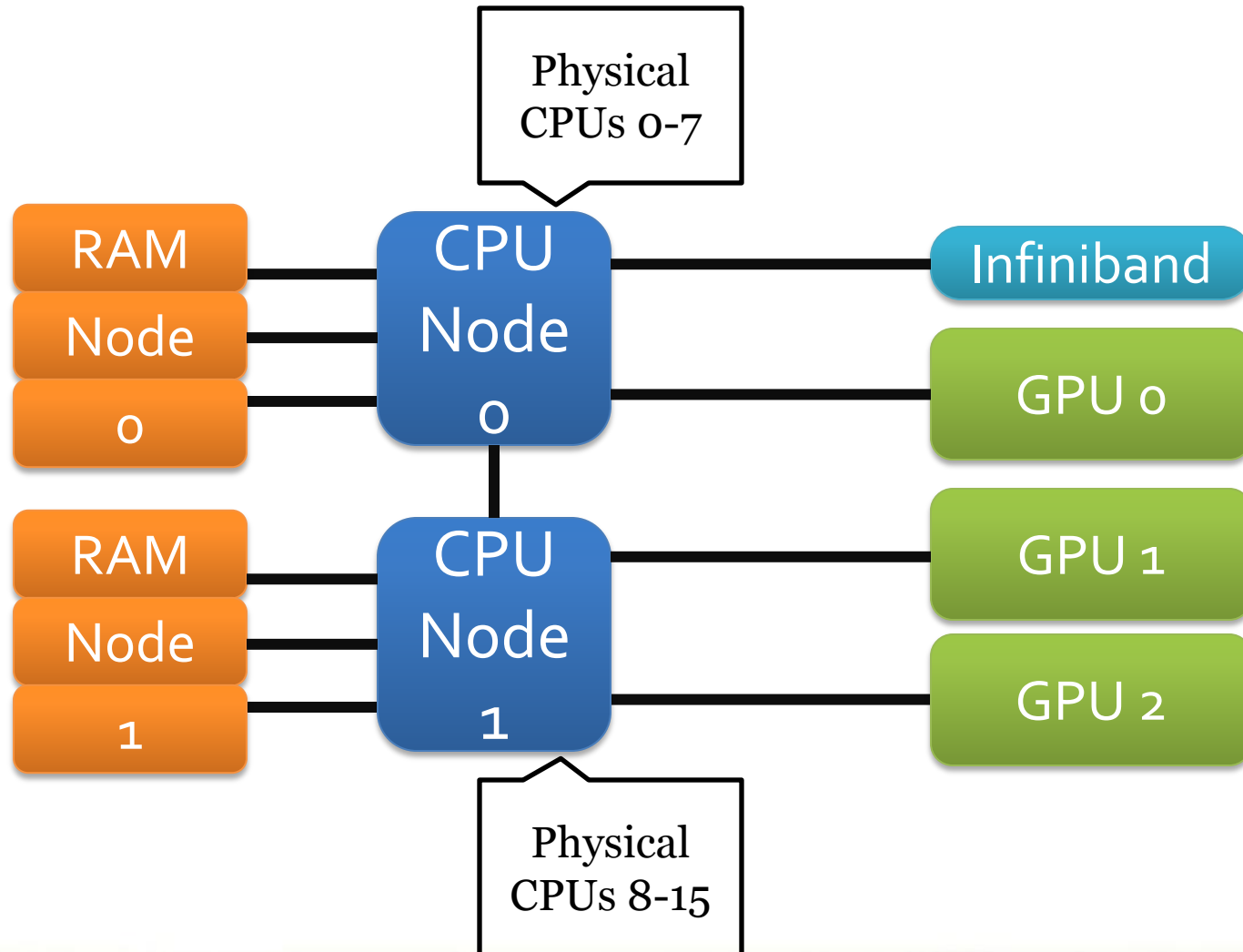


Thread Splitting

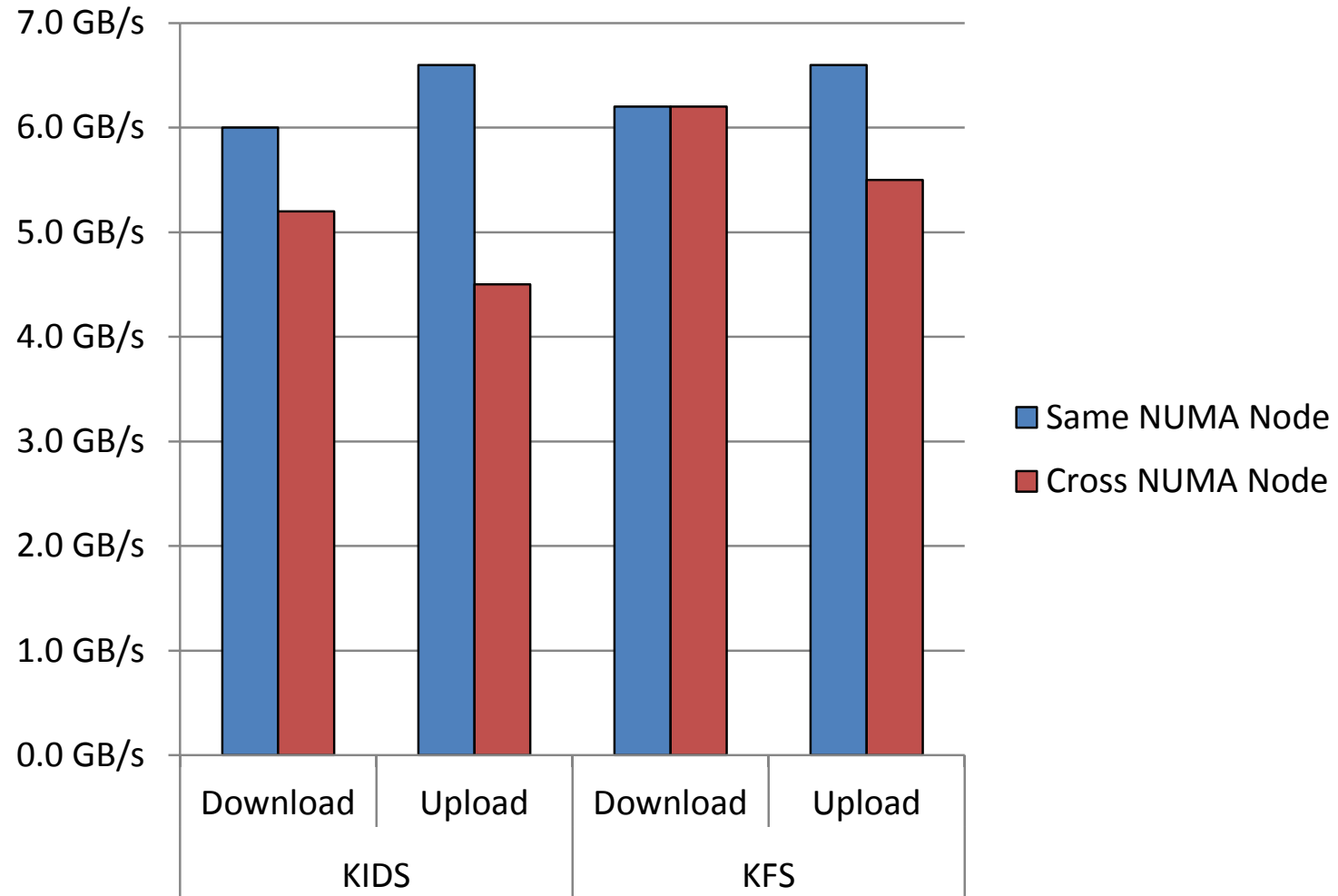
- Instead of 1 thread that controls a GPU and issues MPI calls, split into two threads and bind to appropriate CPU sockets



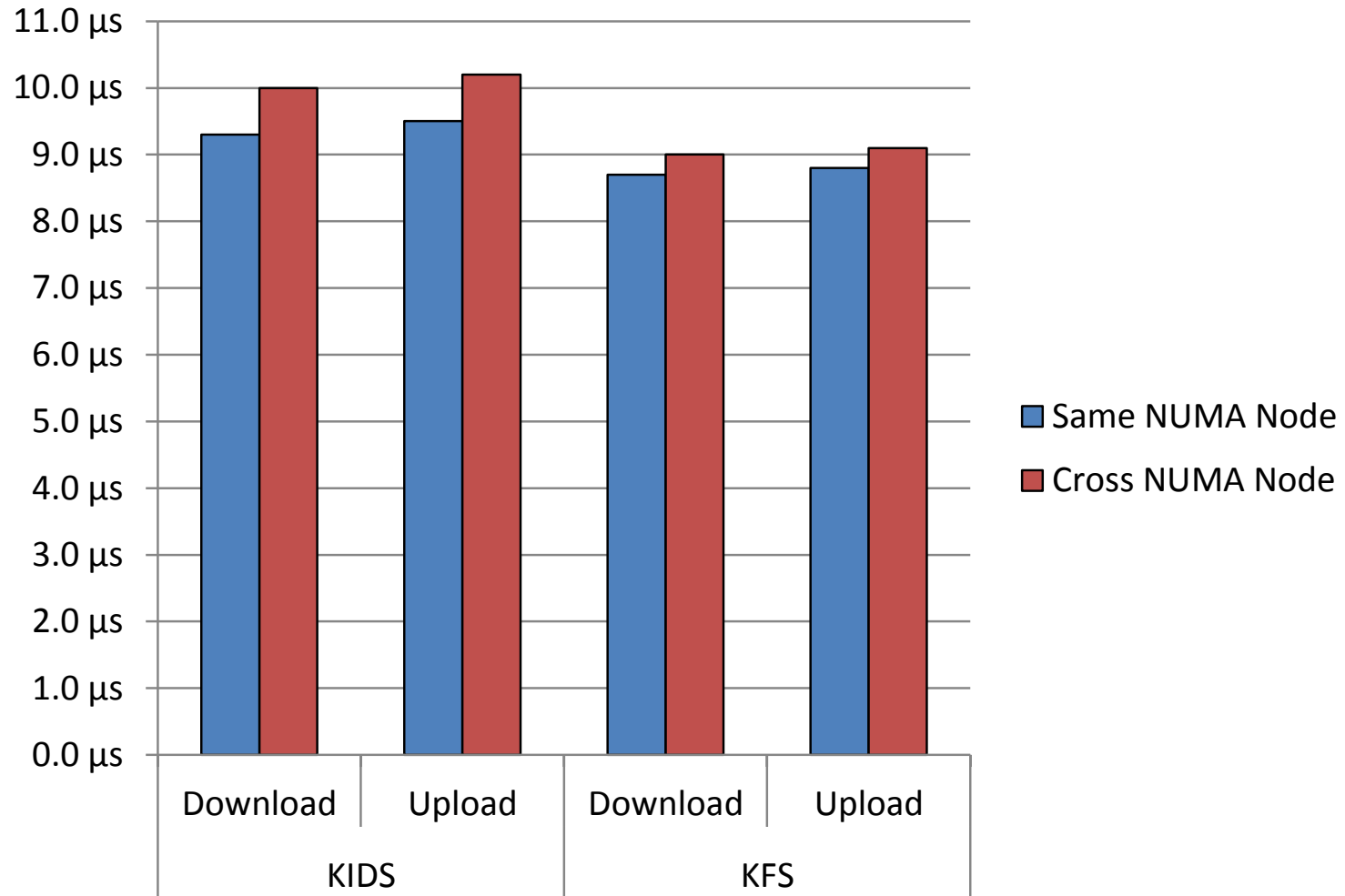
“NUMA Nodes” on KFS nodes



KIDS/KFS NUMA Penalty: OpenCL Bandwidth



KIDS/KFS NUMA Penalty: OpenCL Latency



KIDS/KFS GPU Transfer Performance

- New Sandy Bridge CPUs (on KFS) have PCIe directly attached
- Bandwidth
 - Absolute performance is similar
 - Download NUMA penalty virtually eliminated
 - Upload NUMA penalty somewhat reduced
- Latency
 - Absolute latency improves
 - Download/upload NUMA penalty reduced

GPU DIRECT

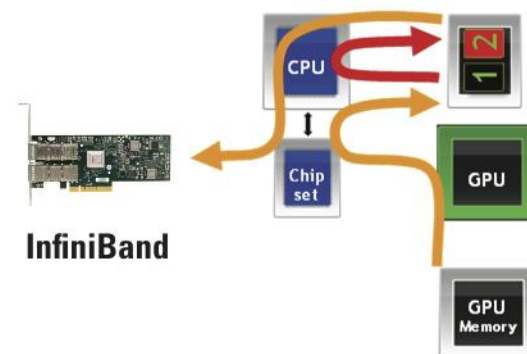


GPU Direct

- Transferring data between GPUs in a scalable heterogeneous system like KIDS is expensive
 - Between GPUs in different nodes
 - Between GPUs in the same node

The Problem with Inter-Node Transfers

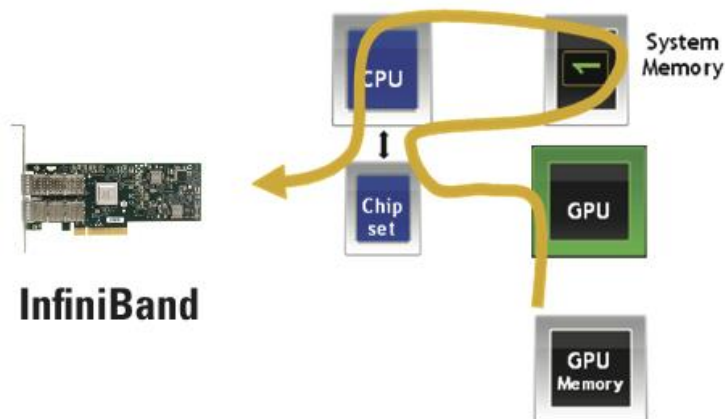
- Data is in device memory of GPU on one node, needs to be transferred to device memory of GPU on another node
- Several hops:
 - Data transferred from GPU memory to GPU buffer in host memory
 - Data copied from GPU buffer to IB buffer in host memory
 - Data read by IB HCA using RDMA transfer
 - Repeat in reverse on other end



http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf

GPUDirect

- NVIDIA and Mellanox developed an approach for allowing others to access the GPU buffer in host memory
- Eliminates the data copy from GPU buffer to IB buffer
 - Eliminates two system memory data copy operations (one on each end)
 - Keeps host CPU out of the data path
 - Up to 30% performance improvement (according to NVIDIA)

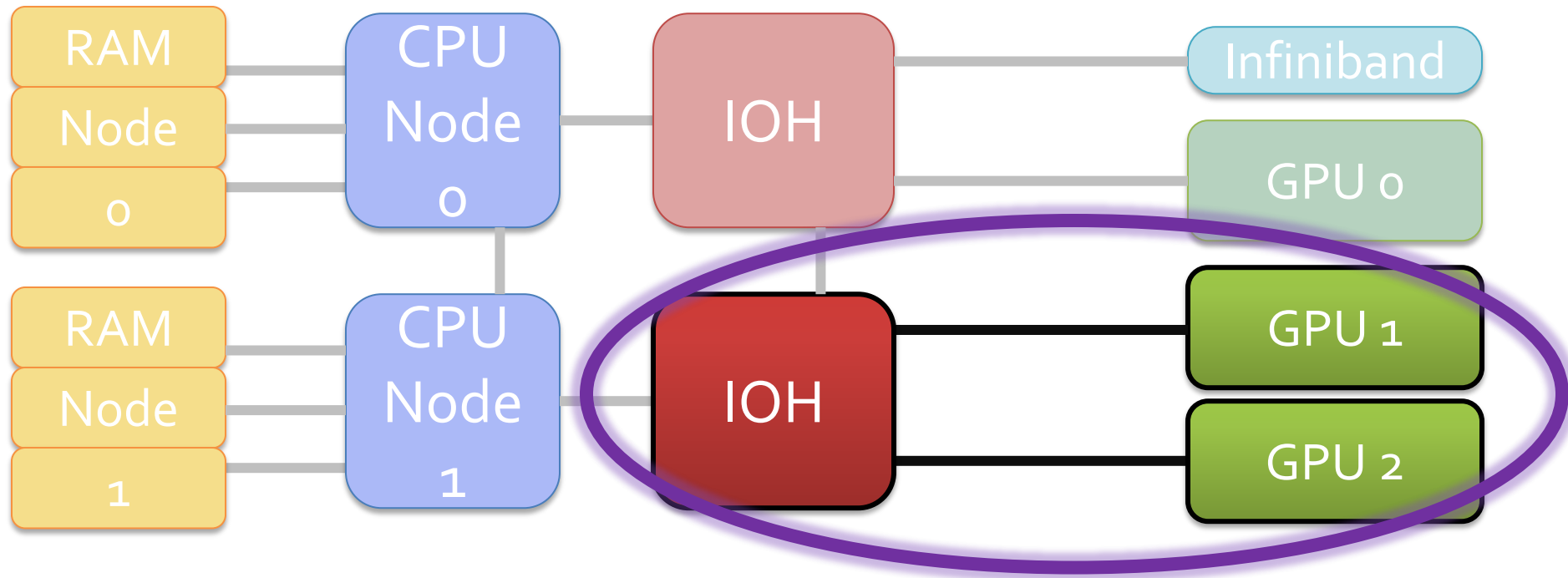


http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf

GPUDirect 2.0: Improving Transfer Performance Within a Node

- Similar problem when transferring data from one GPU to another within the same node
- Old way:
 - Copy data from GPU 1 to host memory
 - Copy data from host memory to GPU 2
- New way:
 - Copy data from GPU 1 to GPU2 without host CPU involvement
- Integrates well with Unified Virtual Addressing feature (single address space for CPU and 1+ GPUs)
- Available since CUDA 4.0

Current GPUDirect support on KIDS



- Currently active on KIDS for GPU1 \leftrightarrow GPU2
 - 2.8 GB/s normally, 4.9 GB/s with GPUDirect

Using GPUDirect

- General strategy:
 - GPU-GPU copies
 - Use cudaMemcpy with two device pointers
 - Enable peer access in CUDA to allow direct GPU-GPU
 - even allows inter-GPU access within CUDA kernels
 - Host-device copies
 - Allocated any host memory as pinned in CUDA
 - CUDA driver puts this in user-pageable memory, virtual address space
 - May need to “export CUDA_NIC_INTEROP=1” for InfiniBand to share this with CUDA

Checking GPUDirect for GPU1 ↔ GPU2

1. Are devices using Tesla Compute Cluster driver?

- `cudaDeviceProp prop1, prop2;`
- `cudaGetDeviceProperties(&prop1, 1);`
- `cudaGetDeviceProperties(&prop2, 2);`
- *check* `prop1.tccDriver==1` *and* `prop2.tccDriver==1`

2. Do devices support peer access to each other?

- `int access2from1, access1from2;`
- `cudaDeviceCanAccessPeer(&access2from1, 1, 2);`
- `cudaDeviceCanAccessPeer(&access1from2, 2, 1);`
- *check* `access2from1==1` *and* `access1from2==1`

Enabling GPUDirect for GPU1 \Leftrightarrow GPU2

3. Enable device peer access both directions:

- `cudaSetDevice(1);`
- `cudaDeviceEnablePeerAccess(2, flags); //flags=0`
- `cudaSetDevice(2);`
- `cudaDeviceEnablePeerAccess(1, flags); //flags=0`

4. Example: send data directly from GPU2 to GPU1:

- `float *gpu1data, *gpu2data;`
- `cudaSetDevice(1);`
- `cudaMalloc(&gpu1data, nbytes);`
- `cudaSetDevice(2);`
- `cudaMalloc(&gpu2data, nbytes);`
- `cudaMemcpy(gpu1data, gpu2data, cudaMemcpyDefault);`

MPI AND GPU TASK MAPPING

How to combine GPUs and MPI?

- **Use 1 MPI task per CPU core?**
 - Simplest for an existing MPI code
 - particularly if they are not threaded
 - Either time share GPUs ...
 - performance can vary, especially with more tasks/GPU
 - ... or only use GPUs from some MPI tasks
 - introduce load balance problem

How to combine GPUs and MPI?

- **Use 1 MPI task per GPU? Per CPU socket?**
 - thread/OpenMP/OpenCL to use more CPU cores
 - ratios like 3GPU:2CPU add complexity
 - pinning 3 tasks to 2 CPU sockets makes using 12 cores hard
 - optimal NUMA mapping may not be obvious
 - can use 1 task for 2 GPUs, leave 3rd GPU idle
 - with 2 I/O hubs, bandwidth is probably sufficient
 - can leave CPU cores idle
 - for codes that match GPUs well, this can be a win
 - recent NVIDIA HPL results show benefits of this approach

How to combine GPUs and MPI?

- **Use 1 MPI task per compute node?**
 - With work, can be highly optimized:
 - Best use of GPUDirect transfers (GPU-GPU, GPU-NIC)
 - Can use numactl library within the task
 - Very complex – must handle:
 - multiple GPUs in one task
 - offload work for all CPU cores
 - NUMA mapping is a challenge
 - especially for automated threading like OpenMP

Bonus Slides